ЯЗЫКИ ПРОГРАММИРОВАНИЯ

РАЗРАБОТКА И РЕАЛИЗАЦИЯ

4-Е ИЗДАНИЕ

Т. ПРАТТ, М. ЗЕЛКОВИЦ

СЕРИЯ

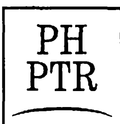
КЛАССИКА COMPUTER SCIENCE

PROGRAMMING LANGUAGES

DESIGN AND IMPLEMENTATION

Fourth Edition

Terrence W. Pratt
Marvin V. Zelkowitz



Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com

КЛАССИКА COMPUTER SCIENCE

Т. ПРАТТ, М. ЗЕЛКОВИЦ

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

РАЗРАБОТКА И РЕАЛИЗАЦИЯ

4-Е ИЗДАНИЕ

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2002

Теренс Пратт, Марвин Зелкович

Языки программирования: разработка и реализация. 4-е издание

Под общей редакцией А. Матросова

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*А. Васильев*
Научный редактор	*А. Матросов*
Переводчик	*А.Михайлова*
Литературный редактор	*А. Телов*
Художник	*Н. Биржаков*
Иллюстрации	*М. Жданова*
Корректоры	*М. Одинокова, Н. Рощина*
Верстка	*А. Келле-Пелле*

ББК 32.973-018.1

УДК 681.3.06

Пратт Т., Зелкович М.

П70 Языки программирования: разработка и реализация / Под общей ред. А. Матросова. — СПб.: Питер, 2002. — 688 с.: ил.

ISBN 5-318-00189-0

В книге известных американских специалистов в области языков программирования Т. Пратта и М. Зелковица рассматриваются общие концепции разработки и реализации языков программирования, а также основы формальных грамматик и конечных автоматов — математических моделей, используемых для определения и реализации языков программирования. Это именно та база, которая необходима высококвалифицированному программисту для создания производительных и устойчивых к ошибкам программ.

Изложение материала в книге не привязано ни к какому конкретному языку программирования, хотя предполагается, что читатель знаком хотя бы с одним процедурным и с одним объектно-ориентированным языком.

Книга будет полезна студентам высших учебных заведений, а также программистам любой квалификации.

© Prentice Hall, Inc., 2001, 1996, 1984, 1975

© Перевод на русский язык, ЗАО Издательский дом «Питер», 2002

© Издание на русском языке, оформление, ЗАО Издательский дом «Питер», 2002

Права на издание получены по соглашению с Prentice Hall

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственность за возможные ошибки, связанные с использованием книги.

ISBN 5-318-00189-0

ISBN 0-13-027678-2 (англ.)

ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Лицензия ИД № 05784 от 07.09.01.
Налоговая льгота – общероссийский классификатор продукции ОК 005-93, том 2; 953005 – литература учебная.
Подписано в печать 20.06.02. Формат 70×100/16. Усл. п. л. 55,47. Тираж 4000 экз. Заказ № 705.
Отпечатано с готовых диапозитивов в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

От издательства 10
Предисловие научного редактора перевода 11
Предисловие к русскому изданию 13
Глава 1. Проблемы разработки языка 16
Глава 2. Влияние машинной архитектуры 64
Глава 3. Вопросы трансляции языка 92
Глава 4. Моделирование свойств языка 142
Глава 5. Элементарные типы данных 181
Глава 6. Инкапсуляция 236
Глава 7. Наследование 303
Глава 8. Управление последовательностью действий 332
Глава 9. Управление подпрограммами 384
Глава 10. Управление памятью 459
Глава 11. Распределенная обработка данных 483
Глава 12. Сетевое программирование 530
Приложение. Обзоры языков 564
Библиография 669
Алфавитный указатель 675

Содержание

От издательства 10
Предисловие научного редактора перевода 11
Предисловие к русскому изданию 13

Глава 1. Проблемы разработки языка 16

1.1. Зачем изучать языки программирования? 16
1.2. Краткая история языков программирования 19
1.2.1. Разработка первых языков 19
1.2.2. Эволюция архитектуры программного обеспечения 23
1.2.3. Области применения 30
1.3. Роль языков программирования 34
1.3.1. Какой язык следует считать хорошим? 37
1.3.2. Парадигмы языка 43
1.3.3. Стандартизация языка 47
1.3.4. Интернационализация 51
1.4. Среда программирования 53
1.4.1. Влияние на разработку языка 53
1.4.2. Среда разработки 56
1.4.3. Языки управления заданиями и языки создания процессов 57
1.5. Обзор языка С 58
1.6. Рекомендуемая литература 61
1.7. Задачи и упражнения 61

Глава 2. Влияние машинной архитектуры 64

2.1. Структура и принципы работы компьютера 64
2.1.1. Аппаратные средства компьютера 65
2.1.2. Программно-аппаратный компьютер 71
2.1.3. Трансляторы и виртуальная архитектура 73
2.2. Виртуальные компьютеры и время связывания 78
2.2.1. Виртуальные компьютеры и реализация языка 79
2.2.2. Иерархия виртуальных компьютеров 80
2.2.3. Связывание и время связывания 82
2.2.4. Обзор языка Java 87
2.3. Рекомендуемая литература 89
2.4. Задачи и упражнения 90

Глава 3. Вопросы трансляции языка 92

3.1. Синтаксис языка программирования 92
3.1.1. Общие синтаксические критерии 93
3.1.2. Синтаксические элементы языка 97
3.1.3. Общая структура программы-подпрограммы 101
3.2. Этапы трансляции 105
3.2.1. Анализ исходной программы 106
3.2.2. Синтез объектной программы 110
3.3. Формальные модели трансляции 112
3.3.1. НФБ-грамматика 114
3.3.2. Конечные автоматы 123

3.3.3. Обзор языка Perl ... 127
3.3.4. Автоматы с магазинной памятью ... 130
3.3.5. Общие алгоритмы грамматического разбора ... 132
3.4. Грамматический разбор на основе метода рекурсивного спуска ... 133
3.5. Обзор языка Pascal ... 135
3.6. Рекомендуемая литература ... 138
3.7. Задачи и упражнения ... 139

Глава 4. Моделирование свойств языка ... 142

4.1. Формальные свойства языков ... 143
4.1.1. Иерархия грамматик Хомского ... 144
4.1.2. Неразрешимость ... 147
4.1.3. Сложность алгоритма ... 153
4.2. Семантика языка ... 156
4.2.1. Атрибутивные грамматики ... 159
4.2.2. Денотационная семантика ... 161
4.2.3. Обзор языка ML ... 168
4.2.4. Проверка правильности программы ... 169
4.2.5. Алгебраические типы данных ... 174
4.3. Рекомендуемая литература ... 177
4.4. Задачи и упражнения ... 177

Глава 5. Элементарные типы данных ... 181

5.1. Свойства типов и объектов ... 181
5.1.1. Объекты данных, переменные и константы ... 181
5.1.2. Типы данных ... 186
5.1.3. Объявления ... 193
5.1.4. Контроль типов и преобразование типов ... 195
5.1.5. Присваивание и инициализация ... 201
5.2. Скалярные типы данных ... 205
5.2.1. Численные типы данных ... 205
5.2.2. Перечисления ... 213
5.2.3. Логические (булевы) значения ... 214
5.2.4. Символы ... 215
5.3. Составные типы данных ... 216
5.3.1. Строки символов ... 217
5.3.2. Указатели и объекты данных, конструируемые программистом ... 220
5.3.3. Файлы и ввод-вывод ... 223
5.4. Обзор языка FORTRAN ... 229
5.5. Рекомендуемая литература ... 231
5.6. Задачи и упражнения ... 232

Глава 6. Инкапсуляция ... 236

6.1. Структурированные типы данных ... 238
6.1.1. Структурированные объекты данных и типы данных ... 238
6.1.2. Спецификация типов структур данных ... 238
6.1.3. Реализация типов структур данных ... 240
6.1.4. Объявления структур данных и контроль типов ... 245
6.1.5. Векторы и массивы ... 246
6.1.6. Записи ... 257
6.1.7. Списки ... 265
6.1.8. Множества ... 268
6.1.9. Выполняемые объекты данных ... 272
6.2. Абстрактные типы данных ... 272
6.2.1. Эволюция понятия типов данных ... 273
6.2.2. Сокрытие информации ... 274
6.3. Инкапсуляция при помощи подпрограмм ... 276
6.3.1. Подпрограммы как абстрактные операции ... 277
6.3.2. Определение и вызов подпрограмм ... 279
6.3.3. Определения подпрограмм как объектов данных ... 284
6.4. Определения типов ... 285
6.4.1. Эквивалентность типов ... 287
6.4.2. Определение типов с параметрами ... 291

6.5. Обзор языка C++ ... 293
6.6. Рекомендуемая литература ... 295
6.7. Задачи и упражнения ... 295

Глава 7. Наследование ... 303

7.1. Повторное рассмотрение абстрактных типов данных ... 304
7.2. Наследование ... 311
7.2.1. Производные классы ... 313
7.2.2. Методы ... 316
7.2.3. Абстрактные классы ... 318
7.2.4. Обзор языка Smalltalk ... 319
7.2.5. Объекты и сообщения ... 321
7.2.6. Концепции абстракций ... 325
7.3. Полиморфизм ... 327
7.4. Рекомендуемая литература ... 330
7.5. Задачи и упражнения ... 331

Глава 8. Управление последовательностью действий ... 332

8.1. Явное и неявное управление последовательностью действий ... 332
8.2. Управление последовательностью действий при вычислении арифметических выражений ... 333
8.2.1. Древовидное представление ... 334
8.2.2. Представление выражений во время выполнения программы ... 343
8.3. Управление последовательностью выполнения операторов ... 348
8.3.1. Базовые операторы ... 348
8.3.2. Структурированное управление последовательностью действий ... 354
8.3.3. Первичные программы ... 363
8.4. Последовательность вычисления неарифметических выражений ... 367
8.4.1. Обзор языка Prolog ... 368
8.4.2. Сопоставление с образцом ... 369
8.4.3. Унификация ... 373
8.4.4. Откат ... 378
8.4.5. Принцип резолюции ... 379
8.5. Рекомендуемая литература ... 381
8.6. Задачи и упражнения ... 381

Глава 9. Управление подпрограммами ... 384

9.1. Управление последовательностью подпрограмм ... 384
9.1.1. Простые подпрограммы вызов-возврат ... 386
9.1.2. Рекурсивные подпрограммы ... 393
9.1.3. Объявление forward в языке Pascal ... 395
9.2. Атрибуты управления данными ... 397
9.2.1. Имена и среды ссылок ... 398
9.2.2. Статическая и динамическая области видимости ... 404
9.2.3. Блочная структура ... 407
9.2.4. Локальные данные и среды локальных ссылок ... 409
9.3. Передача параметров ... 415
9.3.1. Фактические и формальные параметры ... 416
9.3.2. Методы передачи параметров ... 418
9.3.3. Семантика передачи параметров ... 422
9.3.4. Реализация передачи параметров ... 424
9.4. Явно определяемая общая среда ... 435
9.4.1. Динамическая область видимости ... 439
9.4.2. Статическая область видимости и блочная структура ... 443
9.5. Рекомендуемая литература ... 451
9.6. Задачи и упражнения ... 452

Глава 10. Управление памятью ... 459

10.1. Размещаемые в памяти элементы ... 460
10.2. Память, управляемая программистом и системой ... 462
10.3. Статическое управление памятью ... 463

10.4. Управление кучей ... 464
10.4.1. Обзор языка LISP ... 465
10.4.2. Элементы фиксированного размера ... 467
10.4.3. Элементы переменного размера ... 474
10.5. Рекомендуемая литература ... 478
10.6. Задачи и упражнения ... 479

Глава 11. Распределенная обработка данных ... 483

11.1. Различные варианты управления подпрограммами ... 483
11.1.1. Исключения и обработчики исключений ... 484
11.1.2. Сопрограммы ... 489
11.1.3. Планируемые подпрограммы ... 491
11.2. Параллельное программирование ... 493
11.2.1. Параллельное выполнение ... 495
11.2.2. Охраняемые команды ... 496
11.2.3. Обзор языка Ada ... 499
11.2.4. Задачи ... 502
11.2.5. Синхронизация задач ... 504
11.3. Развитие аппаратной части компьютера ... 516
11.3.1. Конструирование процессоров ... 517
11.3.2. Конструирование систем ... 521
11.4. Архитектура программного обеспечения ... 523
11.4.1. Сохраняемые данные и системы транзакций ... 523
11.4.2. Сети и клиент-серверные вычисления ... 525
11.5. Рекомендуемая литература ... 527
11.6. Задачи и упражнения ... 527

Глава 12. Сетевое программирование ... 530

12.1. Настольные издательские системы ... 532
12.1.1. Подготовка документов в LaTeX ... 532
12.1.2. WYSIWIG-редакторы ... 535
12.1.3. Postscript ... 536
12.1.4. Виртуальная машина Postscript ... 536
12.2. Всемирная паутина WWW ... 542
12.2.1. Интернет ... 542
12.2.2. Сценарии CGI ... 555
12.2.3. Апплеты Java ... 558
12.2.4. XML ... 561
12.3. Рекомендуемая литература ... 562
12.4. Задачи и упражнения ... 562

Приложение. Обзоры языков ... 564

П.1. Ada ... 564
П.2. C ... 581
П.3. C++ ... 592
П.4. FORTRAN ... 601
П.5. JAVA ... 611
П.6. LISP ... 616
П.7. ML ... 624
П.8. Pascal ... 635
П.9. Perl ... 645
П.10. Postscript ... 649
П.11. Prolog ... 653
П.12. Smalltalk ... 659
П.13. Рекомендуемая литература ... 667

Библиография ... 669
Алфавитный указатель ... 675

От издательства

Время сжимается под прессом цивилизации. Для того, чтобы Платона признали классиком, потребовались века. Диккенса — столетие. Авторы же произведений, которые будут выходить в нашей серии «Классика computer science», — в большинстве своем достаточно далекие от старости люди. При этом вряд ли кто-нибудь сможет возразить против причисления Теренса Пратта, Эндрю Таненбаума и Брюса Шнайера к настоящим классикам компьютерной литературы.

Возможно, название серии будет резать слух ревнителям чистоты языка, но они должны понять нас: мы оказались в ситуации, когда, как писал Габриель Маркес, «еще не все вещи имеют свои имена». Русский эквивалент фундаментальной, основополагающей части всех компьютерных и информационных технологий, называемой «computer science», пока не найден. Кроме того, помещение в название серии слов из разных языков символично отражает национальную пестроту и разноязычие всего компьютерного сообщества.

Большинство книг компьютерной литературы устаревает, не дождавшись обшарпывания обложки: стремительные смены технологий ставят крест на их актуальности. И лишь некоторые из них известны всем поколениям — от зубров программирования до делающих первые шаги «ламеров». Они переиздаются из года в год и над их текстами постоянно идет работа. Фундаментальность, научность и непреходящая актуальность стали для них замечательным средством от старения. И еще — простой и понятный стиль изложения, ибо, перефразируя Воннегута, «шарлатан тот, кто не может простыми словами объяснять, чем занимается».

В работе над книгами серии «Классика computer science» заняты лучшие силы компьютерной редакции издательства, и мы надеемся, что их труд будет затрачен не напрасно.

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На web-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Руководитель проекта А. Васильев

Предисловие научного редактора перевода

В книге известных американских специалистов в области языков программирования Т. Пратта и М. Зелковица рассматриваются общие концепции разработки и реализации языков программирования. В ней также изложены основы формальных грамматик и конечных автоматов — математических моделей, используемых для определения и реализации языков программирования. Это именно те вопросы, которыми обязан владеть высококвалифицированный программист, так как их знание и понимание позволяют ему создавать более производительные и устойчивые к ошибкам программы.

В США эта книга выдержала четыре издания (первое появилось в 1975 г.). Столь долгая жизнь книги объясняется просто — каждое новое издание отражало появлявшиеся в период между ними новые идеи и концепции обработки информации, находившие свое отражение в свойствах соответствующих языков программирования, причем в этом отношении показателен интервал между выходом очередных изданий книги: если первые три издания появлялись приблизительно с десятилетним перерывом, то между третьим и четвертым прошло всего пять лет.

Перевод первого издания книги (тогда только одного автора — Т. Пратта), осуществленный в 1979 г. издательством «Мир», хорошо известен российским специалистам, которые в конце 70-х — начале 80-х гг. прошлого столетия работали в области компьютерных технологий. Следует отметить, что на основе этой книги были разработаны учебные курсы по общей теории языков программирования и методов трансляции не только в американских университетах, но и в отечественных вузах, что способствовало подготовке высококвалифицированных специалистов в области информационных технологий.

К сожалению, в силу ряда объективных причин уровень подготовки специалистов в области информационных технологий в нашей стране резко упал, в том числе и в связи с отсутствием учебной литературы, которая бы отражала изменения, происходящие в области разработки языков программирования. В сегодняшних университетах студентам чаще преподают навыки программирования на конкретных языках программирования (то есть их синтаксис), а общие вопросы семантики языков программирования и проблемы их разработки и реализации остаются в стороне. Появилось целое поколение молодых программистов, не имеющих представления ни о математических моделях, лежащих в основании языков программирования, ни о методах и способах их реализации.

Появление перевода очередного издания книги Т. Пратта и М. Зелковица, в которой отражено влияние самых последних новаций в области информационных

технологий на идеологию языков программирования — широко используемой в настоящее время объектно-ориентированной парадигмы, бо́льшую ориентацию программных систем на обработку документов, распределенное и сетевое программирование, включая Интернет, и др., — сможет восполнить существующий в отечественной учебной литературе дефицит добротных книг из области общей теории языков программирования и разработки их трансляторов.

Изложение материала в книге не привязано к какому-то конкретному языку программирования, хотя предполагается, что читатель знаком хотя бы с одним процедурным и с одним объектно-ориентированным языком. Для иллюстрации обсуждаемых свойств используются конструкции того языка программирования, в котором эти свойства реализованы в наиболее полной степени. В приложении, составляющем одну шестую объема всей книги, приведено краткое описание двенадцати наиболее распространенных в настоящее время языков программирования, конструкции которых и используются на протяжении всей книги в качестве иллюстраций к излагаемым свойствам языков программирования. Так что в случае если читатель не знаком с синтаксисом какого-либо языка, он всегда может обратиться к приложению для первоначального знакомства с неизвестным ему языком программирования.

На основе этой книги может быть разработан хороший учебный курс по языкам программирования и методам трансляции, тем более что каждая глава завершается большим количеством упражнений. Каждый программист просто обязан ознакомиться с материалом этой книги, если он желает повысить свой рейтинг и продвинуться по служебной лестнице.

В заключение стоит отметить, что в русском издании устранены не только те неточности и опечатки, список которых был любезно предоставлен авторами, но и те, что встретились при переводе и редактировании книги.

А. Матросов

Предисловие к русскому изданию

Русский перевод четвертого издания книги «Programming Languages: Design and Implementation» продолжает традицию, начатую в предыдущих изданиях, — при описании языков программирования основное внимание уделяется программным и аппаратным архитектурам, необходимым для выполнения программ, написанных на этих языках. Наличие этой информации помогает программисту создавать более эффективные программы и уменьшает количество ошибок. По сравнению с третьим изданием мы усовершенствовали этот подход, а также внесли ряд улучшений в представление лежащих в основе рассматриваемых архитектур теоретических и формальных моделей, формирующих базис для решений, принимаемых при разработке этих языков. В настоящем издании исправлены некоторые ошибки, обнаруженные в исходной английской версии книги.

Языки программирования рождаются, стареют и умирают, а вопросы их архитектуры по-прежнему активно обсуждаются в среде специалистов по информационным технологиям. В 4-м издании представлены важнейшие языки начала XXI века. К числу языков, описанных в 3-м издании, добавились Postscript, Java, HTML и Perl, что обусловлено растущей популярностью программирования для World Wide Web. Pascal, FORTRAN и Ada отошли на второй план, поскольку эти языки постепенно отмирают; вероятно, они будут исключены из будущих изданий книги.

Структура книги соответствует учебному курсу, который более 30 лет преподается в университете штата Мэриленд. При чтении этого курса предполагается, что студент уже знаком с C, Java или C++, поэтому особое внимание уделяется языкам Smalltalk, ML, Prolog и LISP, а также углубленному изучению различных аспектов реализации C++. Изучение C++ расширяет знания студентов в области процедурных языков с дополнениями в виде объектно-ориентированных классов, а знакомство с LISP, Prolog и ML позволяет провести сравнительный анализ разных парадигм программирования. Один или два из этих языков можно заменить на FORTRAN, Ada или Pascal.

Предполагается, что читатель знает хотя бы один процедурный язык — обычно это C, C++, Java или FORTRAN. В вводных главах 1 и 2 приведен обзор материала, необходимого для понимания последующих глав. Глава 1 содержит общие сведения о языках программирования, а в главе 2 дается краткий обзор аппаратной архитектуры компьютера, на котором программы выполняются.

Главную тему книги составляют вопросы разработки и реализации языков программирования. Основа курса закладывается в главах 3 и 5–12. В них описываются базовая грамматическая модель языков программирования и компиляторов (глава 3), элементарные типы данных (глава 5), структуры данных и инкапсуляция

(глава 6), наследование (глава 7), операторы (глава 8), вызов процедур (глава 9), управление памятью (глава 10), распределенная обработка (глава 11) и сетевое программирование (глава 12) — важнейшие аспекты проектирования языков.

В главе 4 углубленно рассматривается семантика языков программирования с введением в верификацию программ, денотационную семантику и λ-исчисление. Студенты первого и второго курсов могут пропустить эту главу. В настоящее издание, как и во все предыдущие, включено большое приложение с кратким описанием 12 языков, более или менее подробно рассматриваемых в книге.

В книге рассматривается 12 тем, рекомендованных в 1991 г. ассоциацией ACM/IEEE Computer Society Joint Curriculum Task Force для изучения в области языков программирования.

Курс по написанию компиляторов одно время занимал центральное место в учебных планах обучения студентов информационных специализаций, но сейчас многие полагают, что не нужно требовать от каждого студента, чтобы он мог написать компилятор, — подобные задачи лучше предоставить высококлассным специалистам, а освободившееся время потратить на такие курсы, как технология программирования, разработка баз данных или другие, отражающие аспекты практического применения информационных технологий. Тем не менее мы полагаем, что навыки проектирования компиляторов должны входить в базовую подготовку каждого хорошего программиста. Поэтому в этой книге особое внимание уделяется компиляции различных языковых структур, а в главе 3 довольно подробно рассматриваются вопросы лексического анализа.

Примеры, приведенные в 12 главах этой книги, написаны на языках FORTRAN, Ada, C, Java, Pascal, ML, LISP, Perl, Postscript, Prolog, C++ и Smalltalk. По мере необходимости приводятся дополнительные примеры на языках HTML, PL/1, SNOBOL4, APL, BASIC и COBOL. Мы постарались привести примеры из широкого круга языков, чтобы преподаватель сам выбрал языки для примеров в своем курсе.

Хотя все представленные языки можно кратко рассмотреть в течение одного семестра, мы не рекомендуем включать в учебный курс практические задачи по каждому из этих языков. Для одного курса подобное разнообразие окажется чрезмерным. Десять программ, написанных на разных языках программирования, не дадут студенту глубоких знаний ни по одному из этих языков. Преподавателю рекомендуется выбрать три или четыре языка и уделить им особое внимание.

Все примеры в книге, за исключением простейших, были протестированы в соответствующих трансляторах. Тем не менее, как четко оговорено в разделе 1.3.3, правильная работа программ в нашей локальной системе еще не гарантирует, что транслятор обрабатывает эти программы в соответствии со стандартами языка. Согласно закону Мэрфи, даже самый простой пример может содержать ошибки. В этом случае мы приносим извинения за все причиненные неудобства.

В завершение мы перечислим основные цели, которыми мы руководствовались при подготовке четвертого издания:

- представить обзор основных парадигм, используемых при разработке современных языков программирования;
- уделить особое внимание нескольким языкам и описать их на таком уровне, чтобы возможности этих языков можно было продемонстрировать на практических примерах;

- исследовать реализацию каждого языка до такой степени, чтобы программист получил представление о соответствии между исходным текстом программы и ее поведением при выполнении;
- руководствуясь формальной теорией, показать, какое место архитектура языков программирования занимает в научно-исследовательских разработках в области информационных технологий;
- предоставить набор задач и ссылки на дополнительные источники информации, чтобы студенты могли самостоятельно расширить свои знания в этой важной области.

Мы с благодарностью принимаем ценные замечания, полученные от читателей 3-го издания, от редакторов 4-го издания и от сотен студентов университета штата Мэриленд, прослушавших наш курс, — их мнение по поводу изложения материала, содержащегося в книге, было особенно ценным для нас.

Изменения в 4-м издании. Читатели 3-го издания найдут в настоящем издании следующие основные изменения.

1. В книге появилась новая глава, посвященная World Wide Web. Язык Java занял свое место среди основных языков программирования, а включение обзора языков HTML и Postscript поможет читателю отойти от классических представлений о языках как о «пожирателях чисел» в стиле FORTRAN.
2. Описание объектно-ориентированной архитектуры переместилось ближе к началу книги, поскольку оно занимает очень важное место в разработке современного программного обеспечения. Многие второстепенные разделы также были перемещены, благодаря чему изложение материала стало более логичным и последовательным.
3. Подробные описания языков во второй части 3-го издания принесли меньше пользы, чем мы рассчитывали. В каждую из глав, в которой отражены основные особенности какого-то из 12 использованных языков, были добавлены разделы с описанием их краткой истории, а описания языков переместились в приложение. Несмотря на все добавления, объем книги практически не увеличился, поскольку из нее был исключен устаревший материал.

Надеемся, что в переводе на русский язык наша книга будет воспринята так же благожелательно, как и в оригинале, и что читатель одобрит наш подход к изложению материала.

Теренс Пратт, Ховардсвилл, штат Вирджиния,
Марвин Зелковиц, Колледж-Парк, штат Мэриленд,
ноябрь 2001 г.

Глава 1. Проблемы разработки языка

Любую систему обозначений, пригодную для описания алгоритмов и структур данных, можно назвать языком программирования; в настоящей книге, однако, речь пойдет лишь о тех языках, которые реализованы и используются при программировании на компьютерах. В каком смысле реализуется язык программирования, мы обсудим в следующих двух главах. Остальные главы посвящены подробному описанию разработки и реализации различных компонентов языка. Нашей задачей является рассмотрение основных языковых концепций, общих для всех без исключения языков программирования, и иллюстрирование их примерами из широкого спектра наиболее распространенных языков.

В этой книге мы будем изучать применение этих языковых концепций на примере двенадцати основных языков программирования: Ada, C, C++, FORTRAN, Java, LISP, ML, Pascal, Perl, Postscript, Prolog и Smalltalk. Кроме того, вкратце мы рассмотрим и некоторые другие языки, оказавшие влияние на интересующую нас область человеческой деятельности. Среди них APL, BASIC, COBOL, Forth, PL/1 и SNOBOL4. Однако, прежде чем приступать к изучению языков программирования с общих позиций, следует разобраться, какую пользу это может принести программисту.

1.1. Зачем изучать языки программирования?

К настоящему моменту разработаны и реализованы сотни языков программирования. Еще в 1969 г. Саммет (Sammet) [95] привел список из 120 достаточно широко используемых в то время языков, и с тех пор их число значительно возросло. Однако большинство программистов пользуются всего несколькими языками, многие же вообще ограничиваются всего одним или двумя. Обычно программисты работают с вычислительными системами, на которых используется какой-либо один определенный язык: Java, C или Ada. Какую же пользу может получить программист от изучения множества различных языков, которыми едва ли станет пользоваться?

На самом деле это имеет смысл, если вы не ограничитесь только поверхностным рассмотрением возможностей языка, а постараетесь постичь его концепции разработки и то, как они влияют на его реализацию. На ум сразу же приходит шесть главных мотивов:

1. *Вы сможете разрабатывать более эффективные алгоритмы.* Многие языки предоставляют возможности, которые при правильном использовании приносят программисту пользу, а при неправильном могут привести к большим затратам компьютерного времени или к логическим ошибкам в программе, на исправление которых потребуется немало сил и времени. Даже программист, пользующийся каким-то определенным языком в течение многих лет, может не понимать всех его возможностей. Типичным примером является *рекурсия* — полезное средство программирования, которое при правильном использовании позволяет осуществлять непосредственную реализацию элегантных и эффективных алгоритмов. Однако неправильное ее использование может привести к катастрофическому увеличению времени выполнения программы. Программист, ничего не знающий о проблемах разработки и трудностях реализации рекурсии, скорее всего, будет избегать этой загадочной конструкции, в то время как элементарное знание ее основных принципов и методов реализации позволит программисту оценить относительную стоимость рекурсии в каждом конкретном языке и определить, оправданно ли ее применение в конкретной ситуации. В литературе постоянно появляются описания новых методов программирования. Для наилучшего использования концепций объектно-ориентированного, логического или параллельного программирования необходимо понимание конкретных языков, в которых эти концепции реализованы. Новые технологии, такие как Интернет (Internet) и Всемирная паутина (World Wide Web), в корне изменили природу программирования. Создание методов программирования, оптимально отвечающих этим новым условиям, требует достаточно глубокого понимания существа языков программирования.
2. *Вы сможете более эффективно использовать тот язык программирования, которым обычно пользуетесь.* Поняв, как в используемом вами языке реализованы те или иные возможности, вы сможете писать значительно более эффективные программы. Например, понимание того, как создаются массивы, строки, списки или записи и как они обрабатываются в данном языке, знание подробностей реализации рекурсии или понимание того, как строятся классы объектов, позволит вам создавать с помощью этих компонентов более эффективные программы.
3. *Вы пополните набор полезных программных конструкций.* Роль языка в мышлении двояка, так как язык одновременно и помогает мышлению, и ограничивает его. Действительно, люди используют язык для выражения мыслей, но язык до такой степени структурирует мышление, что человеку трудно мыслить о чем-то, что не имеет непосредственного словесного выражения. Знание только одного языка программирования налагает подобное ограничение. При поиске способов представления данных и методов их обработки, подходящих для решения некоторой задачи, человек думает только о структурах, которые непосредственно реализованы в знакомом ему языке. Изучая конструкции, имеющиеся в разных языках, программист расширяет свой словарь. Понимание способов реализации конструкций в разных языках особенно важно, поскольку, чтобы использовать некую конструкцию в языке, в котором она не представлена непосредственно, программист должен

осуществить ее реализацию в терминах базовых элементов этого языка. Например, структура управления подпрограммами (subprogram control structure), известная как *сопрограмма*, полезна во многих случаях, но только немногие языки предоставляют ее непосредственную реализацию. Однако программисты, пишущие на языках C или FORTRAN, если они знакомы с концепцией сопрограммы и методами ее реализации, могут легко спроектировать программу, в которой используется сопрограмма, а затем реализовать ее на C или FORTRAN.

4. *Вы сможете выбрать язык программирования*, наиболее подходящий для реализации конкретного проекта. Таким образом уменьшается объем необходимых работ. Приложения, в которых выполняется большой объем численных расчетов, могут быть легко разработаны на таких языках, как C, FORTRAN или Ada. Приложения, используемые для принятия решений, например, в области искусственного интеллекта, лучше всего реализовывать на языках LISP, ML или Prolog. Для Интернет-приложений больше подходят Perl или Java. Знание основных особенностей языка, его достоинств и недостатков дает программисту более широкие возможности выбора.
5. *Вам будет легче изучать новые языки.* Лингвист, глубоко понимающий структуры, лежащие в основе естественных языков, может выучить новый иностранный язык легче и быстрее, чем начинающий, который плохо представляет структуру даже своего родного языка. Также и основательное знание конструкций и способов реализации ряда языков программирования позволяет программисту в случае необходимости быстрее выучить новый язык.
6. *Вам будет легче разрабатывать новые языки.* Большинство программистов даже и не помышляют о разработке нового языка программирования, хотя создаваемые ими приложения в действительности являются некоторой формой языков программирования. Разработчики пользовательского интерфейса для большой программы, такой как, например, текстовый редактор, операционная система или графический пакет, неизбежно сталкиваются со многими из тех проблем, которые встречаются и при разработке языков программирования общего назначения. Многие новые языки основаны на моделях реализации языков C или Pascal. Подобный аспект разработки программ обычно упрощается, если программист хорошо знаком с рядом разнообразных конструкций и способами реализации обычных языков программирования.

Изучение языков программирования есть нечто большее, нежели просто беглое знакомство с их возможностями. В действительности сходство возможностей языков обманчиво. Одинаковые возможности двух различных языков могут быть реализованы совершенно разными способами, и, таким образом, эти две версии могут очень сильно различаться по стоимости использования. Например, почти в каждом языке операция сложения реализована как одна из основных операций, но в различных языках, например в C, COBOL или Smalltalk, стоимость операции сложения может различаться на порядок.

В данной книге рассматривается множество языковых конструкций, которые почти всегда сопровождаются одним или несколькими примерами их реализа-

ции на стандартном компьютере. Однако мы не пытались полностью осветить все возможные способы реализации. Те же самые языки или языковые конструкции при реализации на компьютере читателя могут радикально отличаться по стоимости или по деталям структуры, если используются разные методы реализации или если архитектура компьютера отличается от простой стандартной архитектуры.

1.2. Краткая история языков программирования

С тех пор как в 50-е гг. появились первые языки программирования высокого уровня, методы их разработки и реализации постоянно развиваются. Те 12 языков, которые описаны в данной книге, появились в разное время: первые версии FORTRAN и LISP были разработаны в 50-е гг., история языков Ada, C, Pascal, Prolog и Smalltalk начинается в 70-х гг., в 80-е возникли такие языки, как C++, ML, Perl, Postscript, а язык Java — самый молодой, он появился в 90-х. В 60-е и 70-е гг. новые языки часто возникали при разработке больших проектов по созданию программного обеспечения как их составная часть. Когда в 70-е гг. Министерством обороны США в рамках основного проекта разработки языка Ada был составлен отчет об используемых в то время языках программирования, выяснилось, что в различных оборонных проектах было использовано более 500 языков.

1.2.1. Разработка первых языков

В этом разделе мы в общих чертах опишем языки, разработанные в начале компьютерной эры, начиная с середины 50-х по 70-е гг. Более поздние разработки будут подробно освещены по мере рассмотрения в книге новых языков.

Языки численных расчетов. Первый этап развития компьютерных технологий относится к периоду, который начался до Второй мировой войны и продлился до начала 40-х. Во время Второй мировой войны основной задачей компьютеров, называвшихся *электронными вычислительными устройствами*, было определение баллистических траекторий путем решения дифференциальных уравнений движения.

В начале 50-х начали появляться первые языки, использовавшие символьные системы обозначений. Грейс Хупер (Grace Hooper) возглавил в компании Univac группу, которая разрабатывала язык A-0, а Джон Бэкус (John Backus) создал язык Speedcoding для IBM 701. Оба эти языка предназначались для преобразования простых арифметических выражений в исполняемый машинный код.

Настоящий прорыв произошел в 1957 г., когда Бэкус руководил группой по созданию языка FORTRAN, или транслятора формул (FORmula TRANslator). На ранней стадии разработки FORTRAN был ориентирован на численные вычисления, но конечной целью был вполне законченный язык программирования, включающий в себя управляющие структуры, условные операторы и операторы ввода-вывода. Поскольку немногие верили, что получится язык, способный конкурировать с языком ассемблера, в котором машинные команды кодировались вручную, ос-

новной задачей было создать эффективный исполняемый код, поэтому многие операторы разрабатывались с учетом специфики ЭВМ IBM 704. Концепции языка FORTRAN типа механизма трехветвевого перехода[1] вытекала напрямую из аппаратной архитектуры IBM 407, а операторы типа READ INPUT TAPE сегодня выглядят весьма причудливо. Все это выглядело не очень изящно, но в то время еще не задумывались об *элегантном* программировании, зато разработанный язык позволял писать программы, которые выполнялись достаточно быстро на ЭВМ упомянутого типа.

FORTRAN оказался весьма удачным языком и оставался доминирующим вплоть до 70-х гг. Следующая версия FORTRAN вышла в 1958 г. и стала называться FORTRAN II, а спустя несколько лет появился FORTRAN IV. Поскольку каждый производитель ЭВМ[2] реализовывал для своих компьютеров собственную версию языка, то, понятное дело, царил хаос. В 1966 г. FORTRAN IV стал стандартом под названием FORTRAN 66 и с тех пор дважды подвергался пересмотру, в результате чего возникли стандарты FORTRAN 77 и FORTRAN 90. Наличие большого количества программ, написанных на ранних версиях языка, явилось причиной того, что создаваемые трансляторы должны были удовлетворять требованиям обратной совместимости, что препятствовало внедрению в язык новых идей и концепций программирования.

Поскольку FORTRAN оказался столь успешным языком, в Европе возникли опасения, что IBM будет доминировать в компьютерной отрасли. Немецкое общество прикладной математики (German society of applied mathematics — GAMM) создало комитет по разработке универсального языка. В то же время Association for Computing Machinery (ACM) организовала похожий комитет в США. Несмотря на то что у европейцев было некоторое беспокойство по поводу господства американцев, оба этих комитета слились в один. Под руководством Питера Наура (Peter Naur) этот комитет разработал IAL (International Algorithmic Language). Предлагавшееся название ALGOL (ALGOrithmic Language) было вначале отвергнуто. Но поскольку оно стало общеупотребительным, официальное имя IAL пришлось впоследствии изменить на ALGOL 58. Новая версия появилась в 1960 г., и ALGOL 60 (с небольшими изменениями, сделанными в 1962 г.) с 60-х и до начала 70-х гг. прошлого века был стандартом академического языка программирования.

Если FORTRAN разрабатывался для эффективного использования на IBM 704, перед разработчиками языка ALGOL стояли совершенно другие цели, а именно:

1. Сделать систему обозначений в ALGOL как можно ближе к стандартной математической.
2. ALGOL должен быть приспособлен для описания алгоритмов.
3. Программы на ALGOL должны были компилироваться в машинный язык.
4. ALGOL не должен быть привязан к конкретной машинной архитектуре.

---

[1] Имеется в виду арифметический IF. — *Примеч. науч. ред.*

[2] В те далекие времена в нашей стране было принято компьютеры называть электронными вычислительными машинами (ЭВМ). Термин «компьютер» стал де-факто стандартом с распространением персональных компьютеров. — *Примеч. науч. ред.*

Все эти цели оказались в 1958 г. недостижимы. Чтобы обеспечить независимость от архитектуры, в этот язык не были включены возможности ввода-вывода; для таких операций следовало создавать специальные процедуры. Хотя это и обеспечивало независимость языка от архитектуры компьютера, но в то же время приводило к тому, что каждая реализация неизбежно была несовместима с любой другой. Чтобы сохранить аналогию со строгой математикой, подпрограмма рассматривалась как макроподстановка, что породило концепцию *передачи параметров по имени*; как будет показано в разделе 9.3, эффективно реализовать передачу параметров по имени достаточно трудно.

Хотя ALGOL и имел некоторый успех в Европе, он так и не получил коммерческого успеха в Америке, тем не менее его влияние на другие языки было достаточно велико. В качестве примера приведем разработанную Шварцем (Jules Schwartz) из System Development Corporation (CDS) версию языка IAL — JOVIAL (Jules' Own Version of IAL). Этот язык использовался военно-воздушными силами США для решения прикладных задач.

Бэкус был редактором отчета, в котором определялся язык ALGOL [14]. Он использовал синтаксическую систему обозначений, сопоставимую с концепцией контекстно-свободного языка, разработанной Хомским (Chomsky) [27]. Так произошло внедрение теории формальных грамматик в область языков программирования (раздел 3.3). Поскольку Бэкус и Наур внесли огромный вклад в концепцию разработки ALGOL, эта система обозначений носит их имя — нормальная форма Бэкуса–Наура (НФБ) (Backus–Naur Form — BNF).

Можно привести еще один пример влияния ALGOL. Компания по продаже компьютеров Burroughs, которая после слияния с компанией Sperry Univac образовала компанию Unisys, обнаружила работы польского математика Лукашевича (Lukasiewicz), который разработал методику, позволяющую записывать арифметические выражения без скобок, используя алгоритм вычислений с использованием стека. Эта методика оказала значительное влияние на теорию разработки компиляторов. Используя технологию, основанную на этой методике, компания Burroughs разработала компьютер B5500 со стековой архитектурой и вскоре реализовала компилятор языка ALGOL, значительно превышавший по скорости существовавшие в то время компиляторы языка FORTRAN.

С этого момента ситуация изменяется. В 60-е гг. была разработана концепция пользовательских типов данных, которая не была реализована ни в языке FORTRAN, ни в языке ALGOL. Язык Simula-67, разработанный норвежцами Найгардом (Nygaard) и Далом (Dahl), ввел концепцию классов в ALGOL. В 80-е гг. это натолкнуло Страуструпа (Stroustrup) на идею создать C++ как расширение C с добавлением понятия классов (см. приложение, раздел П.3). В середине 60-х Вирт (Wirth) разработал расширение языка ALGOL — ALGOL-W, пользовавшийся меньшим успехом. Тем не менее в 70-х гг. он же разработал Pascal, который стал языком научного программирования тех лет. Другой комитет, ориентируясь на успех ALGOL 60, разработал язык ALGOL 68, который, однако, оказался слишком сложным для понимания и эффективной реализации.

С началом серийного выпуска в 1963 г. новых компьютеров модели 360 фирма IBM в своей лаборатории Hursley, находящейся в Англии, разработала новый язык — NPL (New Programming Language). В связи с некоторым недовольством

сотрудников Национальной физической лаборатории (English National Physical Laboratory) язык был переименован в MPPL (Multy-Purpose Programming Language). В дальнейшем это название было сокращено до PL/I. Язык PL/I объединил в себе вычислительные возможности языка FORTRAN и возможности бизнес-программирования (обработки деловой информации), имевшиеся в языке COBOL. В 70-е гг. PL/I пользовался некоторой популярностью (так, среди языков, описанных во втором издании этой книги, он обладал наибольшим количеством возможностей), в настоящее же время он практически забыт, поскольку вытеснен такими языками, как C, C++ и Ada. Версия PL/C этого языка в 70-е гг. получила распространение как компилятор PL/I для студентов. BASIC был разработан, чтобы удовлетворить потребность в численных расчетах людей, не имеющих отношения к науке, однако в дальнейшем его возможности были расширены и вышли далеко за рамки первоначальных целей.

Языки обработки деловой информации. Сфера обработки деловой информации стала следующей после численных расчетов областью, которая привлекла внимание разработчиков языков. В 1955 г. группа сотрудников Univac под руководством Грейса Хупера (Grace Hooper) разработала язык FLOWMATIC. Целью было создание приложений для обработки деловой информации с использованием некоторого англоподобного текста. В 1959 г. Министерство обороны США профинансировало совещание по разработке языка CBL (Common Business Language), который должен был стать бизнес-ориентированным языком, максимально использующим английский язык в качестве системы обозначений для своих программ. В связи с несогласованной деятельностью различных компаний для быстрой разработки этого языка был сформирован специальный комитет Short Range Committee. Хотя члены этого комитета думали, что они будут разрабатывать некий промежуточный вариант языка, оказалось, что разработанная ими спецификация, опубликованная в 1960 г., определила новый язык — COBOL (COmmon Business Oriented Language). COBOL пересматривался в 1961 и 1962 гг. и был стандартизован в 1968 г. В 1974 и 1984 гг. он снова подвергся пересмотру. (Дополнительные комментарии можно найти в обзоре языка 8.3.)

Языки искусственного интеллекта. Интерес к языкам искусственного интеллекта возник в 50-е гг., когда компанией Rand Corporation был разработан язык IPL (Information Processing Language). Версия IPL-V стала довольно широко известна, но ее использование ограничивалось тем, что IPL-V не был языком высокого уровня. Огромным шагом вперед стала разработка Джоном Мак-Карти (John McCarthy), сотрудником Массачусетсского технологического института (MIT), языка LISP (LISt Processing) для компьютеров IBM 704. Версия LISP 1.5 стала стандартом для его реализации на многие годы. Развитие LISP продолжается до настоящего времени (см. приложение, раздел П.6).

LISP разрабатывался как функциональный язык обработки списков. Естественной областью приложения LISP явилась разработка стратегии ведения игры, поскольку обычная программа, написанная на языке LISP, могла создавать дерево возможных направлений движения (как связанный список) и, продвигаясь по этому дереву, искать оптимальную стратегию. Другой естественной областью применения этого языка стал автоматический машинный перевод текста, где одна цепочка символов может заменяться на другую. В этой области первой разработкой был

язык COMIT, созданный Ингве (Yngve), сотрудником MIT. Каждый оператор программы, написанной на этом языке, был очень похож на контекстно-независимое правило (context-free production, см. раздел 3.3.1) и представлял собой набор замен, которые можно было осуществить, если в исходных данных обнаруживалась конкретная цепочка символов. Поскольку Ингве запатентовал свой код, группа разработчиков из AT&T Bell Telephone Laboratories решила создать свой собственный язык — так появился SNOBOL (см. обзор языка в разделе 8.4).

Если LISP создавался как язык обработки списков для универсальных приложений, Prolog (см. приложение, раздел П.11) стал специализированным языком с основанными на понятиях математической логики структурами управления и стратегией реализации.

Системные языки. В целях повышения эффективности выполняемых программ использование языка ассемблера в системной области продолжалось достаточно долго даже после того, как для приложений в других областях стали использоваться языки высокого уровня. Многие языки системного программирования, такие как CPL и BCPL, так и не нашли широкого применения. Все изменилось с появлением языка C (см. приложение, раздел П.2). С развитием в 70-е гг. конкурентоспособной среды UNIX, написанной в основном на языке C, была доказана эффективность использования языков высокого уровня и в системной области.

1.2.2. Эволюция архитектуры программного обеспечения

Разработка языков программирования происходит не в вакууме. Так, на конструирование языков огромное влияние оказывает то оборудование, на котором должны выполняться написанные на них программы. Язык как средство решения задачи является лишь одной из составных частей всей используемой технологии. Внешняя среда, поддерживающая выполнение программ, называется *операционной средой*. Среда, в которой программа разрабатывается, кодируется, тестируется и отлаживается, или *среда разработки* (host environment), и операционная среда, в которой программа в конечном счете используется, могут отличаться друг от друга. В настоящее время в компьютерной индустрии наступила третья эра разработки компьютерных программ. Каждая из них оказала сильное влияние на ряд языков, которые в тот момент применялись для разработки приложений.

Эра универсальных ЭВМ

С момента появления первых компьютеров в 40-е и вплоть до 70-х гг. в области вычислений доминировали большие универсальные ЭВМ. Дорогостоящий компьютер занимал тогда огромное помещение и обслуживался большим количеством специалистов.

Пакетные среды. Самая ранняя и простая операционная среда полностью состояла из внешних файлов с данными. Программа брала несколько входных файлов с данными, обрабатывала их и создавала несколько выходных файлов с данными (например, программа начисления заработной платы обрабатывала два входных файла, содержащих основные записи платежных ведомостей и время, отработанное каждым сотрудником в течение недельного платежного периода, а за-

тем выдавала два выходных файла, содержащих обновленные платежные ведомости и платежные чеки). Такая операционная среда называется *средой пакетной обработки* (batch-processing), поскольку входные данные группируются в *пакеты* внутри файлов[1] и в виде пакетов обрабатываются программой. 80-колонная перфокарта, или карта Холлерита (Hollerith card), названная так по имени Германа Холлерита (Herman Hollerith), придумавшего ее для переписи населения США в 1980 г., была неотъемлемой частью компьютеров 60-х гг.

Такие языки, как FORTRAN, COBOL и Pascal, изначально разрабатывались под пакетную среду выполнения, хотя сейчас могут использоваться в интерактивной и встроенной операционной средах.

Интерактивные среды. В начале 70-х гг., ближе к концу эпохи универсальных ЭВМ, появилось интерактивное программирование. Чтобы не использовать при создании программы пачку перфокарт, к компьютеру были подсоединены электронно-лучевые мониторы. В результате исследований, проведенных в 60-е гг. в рамках проекта по разработке компьютера на основе микропроцессоров (MAC — microprocessor-array computer) и операционной системы Multics, осуществлявшегося в MIT, появились компьютеры с возможностью *разделения времени*. В таких системах каждому пользователю выделялись небольшие кванты процессорного времени. Так, например, если каждому пользователю компьютера выделяются кванты в 25 мс и к компьютеру подключено 20 пользователей, то каждый из них за одну секунду может использовать два раза по 25 мс (или 50 мс). Поскольку пользователи зачастую тратят большую часть компьютерного времени на обдумывание задач, а не на непосредственное взаимодействие с компьютером, то те немногие, кто в данный момент реально используют компьютер, имеют возможность пользоваться большим количеством временем, чем отведенная им квота в два отрезка времени в секунду.

При использовании интерактивной среды пользователь общается с программой во время ее выполнения посредством дисплея, на который выводятся выходные данные, и клавиатуры или мыши, позволяющей вводить информацию. Примерами являются системы обработки текстов, электронные таблицы, видеоигры, системы обработки баз данных и системы компьютерного обучения. Наверняка читатель хорошо знаком со всеми приведенными примерами.

Влияние на языки программирования. В языках, разработанных для пакетной среды, файлы обычно являются основой для большинства структур ввода-вывода. Хотя файлы также можно использовать и для интерактивного ввода-вывода на терминал, в этих языках не было необходимости в реализации специальной возможности интерактивного ввода-вывода. Например, файлы обычно хранятся в виде записей фиксированной длины, однако при интерактивном вводе данных программа должна считывать каждый символ по мере его введения с клавиатуры. Также обычно структура ввода-вывода не обеспечивает доступа к специальным устройствам ввода-вывода, используемым во встроенных системах.

---

[1] Здесь под *файлом* понимается картотека или набор подшитых бумаг (основное значение английского слова *file*), а не широко используемый в современной информатике термин для обозначения именованной области связанных данных на внешнем носителе информации (например, диске). Перфокарты чем-то напоминали карточки в библиотечных каталогах, поэтому собранные вместе они также назывались файлом. — *Примеч. науч. ред.*

Ошибки, вызывающие остановку выполнения программы в средах пакетной обработки, допустимы, но они дорого обходятся, поскольку после исправления ошибки программу нужно запускать заново. Также в этих средах программист не имеет возможности немедленно исправить вручную обнаруженную ошибку. Таким образом, при разработке этих языков возможностям обнаружения и обработки ошибок и исключительных ситуаций непосредственно в программе придавалось большое значение. Программа должна была обрабатывать наиболее вероятные ошибки и продолжать свое выполнение без каких-либо остановок.

Третьей отличительной характеристикой среды пакетной обработки является отсутствие временных ограничений на выполнение программы. Обычно в языке, разработанном для такой среды, не заложены средства для отслеживания скорости выполнения или прямого ее контроля.

Операции интерактивного ввода-вывода существенно отличаются от стандартных операций с файлами, которые заложены в большинстве языков, разработанных для пакетных сред обработки. И это затрудняет адаптацию подобных языков программирования к интерактивным средам. Эти различия описаны в разделе 5.3.3. Например, язык С включает в себя функцию доступа к строкам текстового файла, а также функцию, которая выводит на терминал все символы, вводимые пользователем с клавиатуры, в то время как в языке Pascal прямой ввод текста с терминала выглядит очень громоздко. Этим объясняется популярность использования языка С (и разработанного на его основе С++) для написания интерактивных программ.

Также в интерактивных средах используется совершенно другой подход к обработке ошибок. Если пользователь вводит неправильные входные данные с клавиатуры, программа может вывести сообщение об ошибке и предложить произвести исправления. Средства языка для обработки ошибок внутри программы (например, возможность пропустить ошибку и выполнять программу дальше) становятся менее существенными. Хотя (в отличие от сред пакетной обработки) завершение работы программы в связи с возникшей ошибкой, как правило, не применяется.

Часто в интерактивных программах приходится использовать некоторые временные ограничения. Например, в видеоиграх, если пользователь в течение определенного времени не реагирует на выведенную сцену, активизируется некоторая реакция со стороны программы. Если интерактивная программа работает настолько медленно, что не успевает отвечать на вводимую команду в течение разумного времени, она считается непригодной для использования.

Эра персональных компьютеров

Если заглянуть в прошлое, то эра универсальных ЭВМ с разделением времени продлилась очень недолго — с начала 70-х до середины 80-х. На смену им пришли персональные компьютеры.

Персональные компьютеры. 70-е гг. могут быть названы *эрой мини-компьютеров.* Они были значительно меньше и дешевле стандартных универсальных ЭВМ того времени. Аппаратные технологии шли вперед семимильными шагами. Микрокомпьютеры, в которых целый процессор размещался на одной квадратной пластинке из пластика или кремния размером один-два дюйма, с каждым годом ста-

новились все быстрее и дешевле. Стандартные универсальные ЭВМ уменьшились в размерах и из комнаты, заполненной стойками и накопителями на магнитных лентах, превратились в декоративную офисную машину 3–5 футов[1] в длину и 3–4 фута в высоту.

В 1978 г. компания Apple выпустила компьютер Apple II, первый по-настоящему коммерческий персональный компьютер. Он представлял собой небольшую настольную машину, на которой запускался BASIC. Эта машина оказала огромное влияние на рынок образовательных услуг, однако деловой мир скептически отнесся к минимизированному компьютеру Apple.

Состояние дел изменилось в 1981 г. Фирма IBM выпустила свой персональный компьютер, а фирма Lotus разработала свое приложение Lotus 1–2–3, основанное на программе обработки электронных таблиц Visi-Calc. Эта программа стала первой из *прикладных программ-«приманок»* (для расширения круга потенциальных заказчиков), которыми промышленность была вынуждена пользоваться. Именно с этого момента персональные компьютеры стали пользоваться неожиданным прежде успехом.

Началом современной эры персональных компьютеров можно считать январь 1984 г., когда в США проходили матчи на кубок по американскому футболу. Именно во время трансляции матчей на этот кубок по телевидению и была показана реклама компьютера Macintosh фирмы Apple. Он характеризовался оконным графическим пользовательским интерфейсом с мышью для ввода данных. Хотя фирма Xerox разработала эту технологию в своем исследовательском центре PARC (Palo Alto Research Center) раньше, компьютер Macintosh стал первым коммерческим применением данной технологии. Позже внешний вид интерфейса Macintosh был заимствован компанией Microsoft для своей операционной системы Windows и стал основным для персональных компьютеров.

С течением времени компьютеры становились все дешевле и работали быстрее. Компьютер, использовавшийся при написании этой книги, работает в 200–400 раз быстрее, имеет в 200 раз больше памяти, в 3000 раз больше дискового пространства и стоит в три раза дешевле первых персональных компьютеров, цена которых 20 лет назад была $5000. Кроме того, он более мощный, чем те универсальные ЭВМ, которым он пришел на смену.

Среда встроенных систем. Встроенные компьютеры являются боковой ветвью развития персональных компьютеров. *Встроенной компьютерной системой* называется система, которая управляет частью более крупной системы, такой как промышленный завод, станок, автомобиль или даже тостер. Поскольку компьютерная система стала составной частью более крупных систем, то ее неисправность обычно означает сбой в работе всей системы в целом. В отличие от обычных персональных компьютеров, где неудобства от сбоя программы обычно связаны лишь с необходимостью ее перезапуска, сбой встроенного приложения может быть связан с угрозой для человеческой жизни. Из-за сбоя автомобильного компьютера может произойти авария на скоростной трассе. Сбой компьютера в атомной промышленности может вызвать перегрев атомного реактора, а сбой компьютера в больнице может привести к прекращению мониторинга пациентов. В конце концов, из-за

---

[1] 3 фута = 1 ярд = 0,91439841 м. – *Примеч. науч. ред.*

сбоя в ваших электронных часах вы можете опоздать на заседание или встречу. Поэтому к надежности и точности приложений, используемых в таких областях, предъявляются повышенные требования. В этой области широко используются языки Ada, C, C++, поскольку они отвечают специфическим требованиям сред встроенных систем.

Влияние на языки программирования. С появлением персональных компьютеров вновь изменилась роль языка. Во многих прикладных областях производительность перестала быть основным требованием. Компьютер, снабженный таким удобным пользовательским интерфейсом, как многооконный интерфейс Windows, полностью управляется одним пользователем. Благодаря достаточно низким ценам на компьютеры отпала необходимость в разделении времени. Задачей первостепенной важности стала разработка языков с хорошей интерактивной графикой.

В настоящее время основным типом пользовательского интерфейса являются интерфейсы с «оконной» конфигурацией. Пользователи персональных компьютеров, как правило, хорошо знакомы с многооконным интерфейсом: такие понятия, как окно, значок, полоса прокрутки, меню и другие многочисленные элементы управления графического интерфейса стали привычными для абсолютного большинства пользователей. Однако программирование таких пакетов программ может оказаться сложным делом. Поэтому поставщики операционных систем с многооконными пользовательскими интерфейсами создают специальные библиотеки пакетов для работы с окнами. Для облегчения разработки приложений разработчики первым делом должны ознакомиться с этими библиотеками.

Естественной моделью для данной среды является объектно-ориентированное программирование. Использование языков Java и C++ с их иерархией классов облегчает взаимодействие с пакетами, разработанными сторонними производителями.

Программы, написанные для встроенных систем, обычно работают без основной операционной системы и обычной среды взаимодействия с файлами и устройствами ввода-вывода. Наоборот, такие программы должны обращаться к нестандартным устройствам ввода-вывода через специальные процедуры, учитывающие все особенности конкретного устройства. Поэтому в языках, используемых для встроенных систем, файлам и связанным с ними операциям ввода-вывода уделяется меньше внимания. Доступ к специальным устройствам, как правило, осуществляется с использованием тех средств языка программирования, которые обеспечивают доступ к конкретным аппаратным регистрам, областям памяти, обработчикам прерываний или к подпрограммам, написанным на ассемблере или другом языке низкого уровня.

Во встроенных системах особенно важна обработка ошибок. Обычно каждую программу составляют так, чтобы она могла самостоятельно обработать любую ошибку и принять меры для восстановления и продолжения своей работы. Как правило, завершение работы программы при возникновении ошибки не считается допустимым выходом из положения, кроме случаев катастрофического сбоя системы. Кроме того, в таких системах обычно нет пользователя, который мог бы в интерактивном режиме устранить ошибку.

Встроенные системы, как правило, работают в *режиме реального времени*, то есть большая система, в которую интегрирована компьютерная система, требует

от нее ответов на запросы и выдачу выходного сигнала в течение строго определенных интервалов времени. Например, управляющий самолетом компьютер должен мгновенно реагировать на изменение высоты и скорости полета. Для функционирования этих программ в режиме реального времени язык должен поддерживать такие возможности, как отслеживание интервалов времени, реакция на задержку сверх установленного времени (которая может означать сбой некоторой части системы), запуск и завершение определенных действий в назначенное время.

Наконец, встроенная компьютерная система обычно является *распределенной системой*, состоящей из нескольких компьютеров. Программа, работающая в таких распределенных системах, обычно состоит из ряда одновременно выполняемых задач, каждая из которых управляет одной из частей системы или наблюдает за ней. Главная программа, если таковая существует, занимается только запуском этих задач. Будучи однажды запущенными, эти задачи обычно выполняются одновременно и независимо друг от друга, поскольку необходимость в их остановке возникает, только если по какой-либо причине происходит сбой или остановка всей системы.

Сетевая эра

Распределенная обработка данных. По мере того как в 80-е гг. компьютеры становились более быстродействующими, меньше в размерах и дешевле, они начали проникать в деловой мир. В фирме могла иметься главная машина для обработки корпоративных данных, таких как начисление заработной платы, а в каждом отделе мог стоять локальный компьютер для обеспечения технической поддержки — обработки инструкций, написания отчетов и т. д. Для бесперебойной работы организации информация с одного компьютера должна была передаваться для последующей обработки на другой. Например, из отдела сбыта необходимо послать оплаченный заказ в производственный отдел, а финансовому отделу нужна информация по бухгалтерскому учету. Для использования в крупных организациях были разработаны локальные вычислительные сети (ЛВС) с архитектурой *клиент–сервер*, использующие линии телекоммуникаций для связи между компьютерами. Программа-*сервер* должна обеспечивать доступ к информации, а множественные *клиентские* программы могут запрашивать сервер для получения этой информации.

Известным примером приложения клиент–сервер является система бронирования авиабилетов. На мощном компьютере должна находиться база данных расписания авиарейсов. Турагент (или путешественник) запускает клиентскую программу, которая выдает ему информацию о полетах. Если требуется информация по другому рейсу, клиентская программа пошлет запрос серверу на получение или загрузку на клиентский компьютер соответствующей информации. Таким образом, одно серверное приложение может обслуживать множество клиентских программ.

Интернет. В середине 90-х гг. наблюдалось преобразование распределенных ЛВС в международную глобальную сеть Интернет. В 1970 г. DARPA (Defense Advanced Research Projects Agency) начало разработку проекта по соединению универсальных вычислительных машин в большую, надежную и защищенную сеть. Целью ее создания было обеспечение избыточности на случай войны, так чтобы

военные могли иметь доступ к компьютерам из любой точки страны. К счастью, сеть ARPANET не пришлось использовать для этих целей, и с середины 80-х гг. военная сеть ARPANET преобразовалась в ориентированную на исследователей сеть Интернет. Впоследствии в сеть добавлялись новые компьютеры, и в настоящее время во всем мире каждый пользователь может подключить свой компьютер к сети Интернет. Миллионы машин соединены в динамически изменяемый комплекс сетевых серверов.

На заре функционирования глобальной сети Интернет доступ в нее требовал наличия двух типов компьютеров. Пользователь должен был находиться за клиентским персональным компьютером. Для получения информации он должен был подсоединиться к соответствующему серверу. Для этого использовались протоколы telnet и протокол передачи файлов FTP (file transfer protocol). Протокол telnet позволял на компьютере пользователя эмулировать удаленный терминал сервера, что позволяло пользователю непосредственно общаться с удаленным сервером, в то время как протокол FTP просто позволял клиентской машине посылать файлы на сервер или получать их с сервера. В обоих случаях пользователь должен был знать, на какой именно машине находится необходимая ему информация.

В то же самое время был разработан третий протокол — простой протокол передачи сообщений (SMTP — Simple Mail Transfer Protocol). Протокол SMTP — основа сегодняшней электронной почты. Каждый пользователь имеет локальное регистрационное имя на клиентской машине, а каждая машина имеет уникальное собственное имя (например, mvz —регистрационное имя автора этой книги, а aaron.cs.umd.edu — уникальное собственное имя машины, подключенной к Интернету). Послать сообщение можно, используя программу, поддерживающую протокол SMTP, и зная имя пользователя и имя машины, на которой он зарегистрирован (например, mvz@aaron.cs.umd.edu). Здесь важно отметить, что, как правило, нет необходимости знать точный адрес компьютера, на котором зарегистрирован данный пользователь (например, достаточно использовать адрес mvz@cs.umd.edu). Нет никакой необходимости знать точный адрес машины в Интернете.

В конце 80-х основной целью стало упрощение поиска информации в Интернете. Прорыв в этом направлении осуществился в Европейском институте ядерных исследований (CERN), находящемся в Женеве (Швейцария). Бернерс-Ли (Berners-Lee) разработал концепцию *гиперссылок* в рамках языка HTML (HyperText Markup Language) как способа навигации в Интернете. После создания в 1993 г. web-браузера Mosaic и добавления к Интернет-технологиям протокола передачи гипертекстов HTTP (HyperText Transfer Protocol) наконец-то произошло открытие Интернета для широких слоев населения. К концу XX столетия изменилась целая структура поиска информации и получения знаний, так как наличие доступа к Интернету (имеющегося у значительной части населения) позволяет отыскивать необходимые сведения в любом уголке Всемирной паутины.

Влияние на языки программирования. Появление Всемирной паутины (WWW — World Wide Web) вновь изменило роль языков программирования. Вычисления снова стали централизованными, но существенно иным образом, нежели в раннюю эру универсальных компьютеров. По всему миру создаются крупные серверы информационных архивов. Для получения информации пользователи подклю-

чаются к этим серверам через Интернет, а для ее обработки (например, для создания отчета) используют локальные клиентские машины. Вместо того чтобы распространять миллионы копий нового программного обеспечения, поставщик может просто выложить продукт на сайт WWW, а пользователь может загрузить его себе на машину для локального использования. Чтобы пользователь мог загрузить программный продукт, а поставщик программного обеспечения имел возможность получить плату за использование этого продукта, необходим язык программирования, позволяющий вести диалог между клиентским компьютером и сервером. Развитие электронной торговли напрямую зависит от наличия языков с такими возможностями.

Изначально web-страницы были статическими: можно было просмотреть текст, рисунки или графики. Для доступа к другой странице пользователь мог щелкнуть на ее адресе URL (Uniform Resource Locator). Однако для развития электронной коммерции информация должна передаваться в обоих направлениях между клиентской машиной и сервером, поэтому web-страницы должны были стать более активными. Подобные возможности обеспечиваются такими языками программирования, как Perl и Java.

Использование WWW поставило перед языками такие проблемы, которые не были очевидны в предыдущие две эры. Одна из них — безопасность. Посетитель web-сайта должен быть уверен в том, что его владелец не имеет злого умысла и не испортит клиентский компьютер, удалив с него информацию. Эта проблема, характерная для систем с разделением времени, отсутствует для персональных компьютеров, к которым, в принципе, имеет доступ только сам пользователь. Следовательно, доступ со стороны web-сервера к локальным файлам пользователя должен быть ограничен.

Еще одна важная проблема — производительность. Хотя персональные компьютеры стали очень быстродействующими, линии связи, соединяющие пользователя с Интернетом, как правило, имеют ограниченную скорость передачи. Вдобавок, хотя сами машины достаточно быстры, при подключении к серверу достаточно большого количества пользователей он может оказаться перегружен. Чтобы избежать возникновения таких ситуаций, можно обрабатывать информацию на клиентской машине, а не на сервере. Чтобы разгрузить сервер за счет клиентской машины, он должен переслать клиенту небольшую исполняемую программу. Проблема состоит в том, что сервер не знает, каким компьютером является клиентская машина, поэтому не ясно, какого вида должна быть исполняемая программа. Далее мы подробно рассмотрим язык Java, который был специально разработан для решения этой проблемы.

1.2.3. Области применения

Выбор подходящего языка программирования для решения той или иной задачи часто зависит от того, к какой предметной области она принадлежит. Спектр языков программирования, применяемых для решения прикладных задач из различных предметных областей, за последние 30 лет претерпел значительные изменения. В табл. 1.1 подведены итоги использования различных языков программирования в различных предметных областях и в разные периоды.

Таблица 1.1. Использование языков в различных областях

Период	Область применения	Основные языки	Другие языки
1960-е гг.	Обработка деловой информации	COBOL	Assembler
	Научные вычисления	FORTRAN	ALGOL, BASIC, APL
	Системная область	Assembler	JOVIAL, Forth
	Искусственный интеллект	LISP	SNOBOL
Настоящее время	Обработка деловой информации	COBOL, C++, Java, spreadsheet	C, PL/1, 4GLs
	Научные вычисления	FORTRAN, C, C++, Java	BASIC
	Системная область	C, C++, Java	Ada, BASIC, Modula
	Искусственный интеллект	LISP, Prolog	
	Издательская деятельность	TeX, Postscript, текстовые процессоры	
	Создание процессов	UNIX, shell, TCL, Perl, JavaScript	AWK, Marvel, SED
	Новые парадигмы	ML, Smalltalk	Eifell

Приложения 60-х

Программные приложения, которые в 60-е гг. интенсивно разрабатывались, можно разделить на четыре основных типа: обработка деловой информации, научные вычисления, системное программирование, системы искусственного интеллекта.

Приложения для обработки деловой информации. Большинство приложений этого типа, сегодня чаще называемых бизнес-приложениями, было предназначено для обработки огромного количества данных и выполнялось на *больших «железных»* универсальных машинах (*big iron mainframes*). Класс приложений этого типа включал в себя программы для учета поступления заказов, управления ресурсами и персоналом, а также для начисления зарплаты. Они были предназначены для считывания больших объемов данных, собранных за длительный период времени и хранящихся на магнитных лентах, и созданию новых данных, обновленных в результате незначительных преобразований. Чтобы понять, как это выглядело, посмотрите любой научно-фантастический фильм 60-х гг. Крутящиеся бобины с лентами в этих фильмах служили символом *современных компьютерных технологий*.

Для создания таких приложений был разработан язык COBOL. Разработчики этого языка приложили немало усилий, чтобы обеспечить корректную обработку данных. Класс бизнес-приложений включает в себя также деловое планирование, анализ риска и оценку возможных вариантов. В 60-е гг. программисту, использовавшему COBOL, обычно требовалось несколько месяцев на создание типичного приложения оценки возможных вариантов.

Научные вычисления. Приложения этого класса сводятся к нахождению решений различных математических уравнений. Они включают в себя задачи численного анализа, решения дифференциальных и интегральных уравнений и зада-

чи статистики. Для применения именно в этой области изначально и создавались компьютеры — для составления баллистических таблиц во время Второй мировой войны. Здесь всегда доминировал FORTRAN. Его синтаксис всегда был близок к математическому языку, и ученым было легко использовать его.

Системная область. Для создания операционных систем и реализации компиляторов в то время не существовало эффективного языка. Такие приложения должны были иметь доступ ко всем функциональным возможностям и ресурсам аппаратной части компьютера. Для достижения максимальной эффективности часто выбирался язык ассемблер. В некоторых проектах Министерства обороны США использовался JOVIAL — разновидность языка ALGOL, кроме того, вплоть до конца 60-х для таких приложений использовались также языки типа PL/I.

Еще одна близкая область — управление технологическим процессом и управление оборудованием. В связи с дороговизной и большими размерами компьютеров того времени большинство таких приложений (например, программа управления электростанцией или автоматической линией сборки) были больших объемов. Хотя обычно для этих целей использовался ассемблер, для применения в этой области были разработаны такие языки, как Forth.

Искусственный интеллект. Искусственный интеллект в те годы был относительно новой областью исследований, и в разработке приложений в ней доминировал язык LISP. Отличительной особенностью программ, написанных на этом языке, является реализация алгоритмов, осуществляющих поиск в больших объемах данных. Например, при игре в шахматы компьютер генерирует множество потенциальных ходов, а затем в течение отведенного на один ход времени выбирает наилучший вариант.

Приложения XXI века

В свое время язык Ada был разработан с целью устранения дублирования возможностей в конкурирующих между собой языках программирования. В настоящее время ситуация гораздо более сложная, чем в 60-е гг. Сейчас мы имеем больше областей применения, для которых разработаны и реализованы специализированные языки программирования, хорошо адаптированные к решению возникающих в этих областях задач. Более того, в каждой области применения существует несколько подобных языков.

Приложения для обработки деловой информации. По-прежнему основным языком в этой области остается COBOL, хотя иногда используются языки C и C++. Однако сценарий оценки возможных вариантов кардинально изменился. Электронные таблицы, используемые на персональных компьютерах, полностью реформировали эту область применения. В то время как раньше программист тратил несколько месяцев на создание обычной программы делового планирования, теперь аналитик может за несколько часов составить много таких таблиц.

Языки четвертого поколения 4GL (Fourth Generation Languages) также заняли определенную нишу в этой области. Языки 4GL — это языки, специально адаптированные под конкретные области применения обработки деловой информации; как правило, они имеют средства для создания оконного интерфейса и простой доступ к записям базы данных. Также предусмотрены специальные возможности для создания форм заполнения стандартного бланка и генерирования красиво

оформленных отчетов. Иногда компиляторы языков 4GL в качестве результата выдают программы на языке COBOL.

В 1996 г., когда вышло предыдущее издание этой книги, не было даже такого термина, как *электронная коммерция* (*e-commerce*). Это понятие, обозначающее ведение коммерческой деятельности через Интернет (продажа различных товаров через виртуальные магазины, предоставление услуг через электронные офисы), сильно изменило природу бизнес-программирования. Средства, позволяющие вести диалог пользователя (то есть покупателя) и компании (продавца) посредством Всемирной паутины, дали толчок к развитию новой роли языков программирования. Язык Java был разработан для обеспечения конфиденциальности частной жизни пользователя, а такие языки, как Perl и JavaScript позволяют продавцу получить от пользователя сведения, необходимые для проведения сделки.

Научные вычисления. Здесь по-прежнему FORTRAN не сдает своих позиций, хотя языки Java и C++ вполне успешно конкурируют с FORTRAN 90.

Системная область. В этой области доминирует язык C, созданный в конце 60-х гг., и его более новый вариант C++. Язык C обеспечивает очень эффективное выполнение программ и позволяет программисту получить полный доступ к операционной системе и аппаратной части. Кроме того, используются такие языки, как Modula и современный вариант BASIC. Хотя язык Ada и создавался для применения в этой области, он не получил здесь статуса основного языка. Программирование на языке ассемблер стало анахронизмом.

С появлением недорогих микропроцессоров, используемых в автомобилях, микроволновых печах, видеоиграх и электронных часах, возросла необходимость в языках, позволяющих писать программы для работы в реальном времени. К таким языкам относятся C, Ada и C++.

Искусственный интеллект. Здесь по-прежнему используется LISP, хотя на смену MIT LISP 1.5 начала 60-х пришли современные версии Scheme и Common LISP. Также развился Prolog. Оба языка признаны наиболее подходящими для задач поиска оптимального решения.

Издательская деятельность. Издательская деятельность является относительно новой областью применения языков программирования. Системы обработки текстов имеют свой собственный синтаксис входных команд и выходных файлов. Эта книга была написана с помощью системы обработки текстов TEX. Можно сказать (за недостатком более подходящего термина), что главы *компилировались* по мере написания, то есть в них вставлялись ссылки на рисунки и таблицы, размещались сами рисунки и текст разбивался на абзацы.

Транслятор TEX создает программу на языке описания страниц Postscript. Хотя Postscript, как правило, является выходным форматом текстового процессора, у него есть собственный синтаксис и семантика, и текст может быть откомпилирован подходящим процессором. Обычно документы печатаются на лазерном принтере. Некоторые настаивают на программировании непосредственно в Postscript, но на сегодняшний день это выглядит так же глупо, как и программирование на языке ассемблер в начале 60-х (см. раздел 12.1).

Процессы. В 60-е гг. программист активно участвовал в работе компьютеров. Для выполнения какой-либо задачи он должен был дать соответствующую команду, которая затем исполнялась компьютером. Однако в настоящее время для управ-

ления одной программой часто используется другая (например, для регулярного резервного копирования файлов в полночь, ежечасной синхронизации времени, автоматического ответа на электронные письма во время отпуска, автоматического тестирования программы после ее успешной компиляции и т. д.). Такие операции называются *процессами*. В настоящее время имеется значительный интерес к разработке таких языков, в которых можно определять подобные процессы и после успешной трансляции автоматически запускать на выполнение.

В системе UNIX командный язык пользователя называется *командным интерпретатором*, или *оболочкой shell*, а программы называются *сценариями shell*. Эти сценарии активизируются при условии выполнения некоторых допустимых условий. Кроме того, появилось множество других языков сценариев (например, для тех же целей можно использовать Perl или TCL).

Новые парадигмы программирования. Постоянно появляются и изучаются новые модели приложений. В области исследования теории типов в языках программирования используется язык ML. Хотя промышленное применение этого языка не слишком значительно, его популярность постоянно растет. Другой важный язык — Smalltalk. Хотя он также не получил широкого использования в коммерческой области, он оказал глубокое воздействие на идеологию языков. Многие из объектно-ориентированных свойств языков C++ и Ada заимствованы из Smalltalk.

Специализированные языки, предназначенные для решения задач в различных прикладных областях, являются неиссякаемым источником новых исследований и разработок. По мере углубления наших знаний о методах компиляции и способах построения сложных систем постоянно открываются новые области применения языков программирования и возникает необходимость создания новых языков именно для этих областей.

1.3. Роль языков программирования

Изначально языки программирования разрабатывались таким образом, чтобы обеспечить наиболее эффективное выполнение программ. Компьютеры, стоившие миллионы долларов, определяли главную часть расходов, тогда как оплата работы программистов, зарабатывавших примерно $10 000 в год, составляла всего лишь малую их часть. Программа, созданная с помощью любого языка высокого уровня, должна была выполняться по крайней мере так же эффективно, как и программа, написанная на языке ассемблер, где кодирование осуществлялось вручную. Джон Бэкус, в конце 50-х гг. создавший для IBM язык FORTRAN, десятью годами позже говорил так [54]:

«Честно говоря, у нас не было даже отдаленного представления о том, как эта штука [язык и компилятор FORTRAN] будет работать... Поскольку большинство людей в то время совершенно не верили в возможность создания подобной вещи, мы просто делали упор на оптимизацию объектной программы и времени выполнения. Считалось, что машинные коды программ, созданные транслятором, будут совершенно неэффективны и их невозможно будет использовать для большинства приложений.

Мы совершенно не ожидали получить в результате систему, абсолютно независимую от конкретной машины, на которой программа в конечном счете должна выполняться. Оказалось, что это на самом деле очень ценное свойство, но поначалу мы об этом действительно не думали.

Мы никак не организовывали нашу деятельность. Каждую часть программы писали один-два человека, которые, за очень небольшим исключением, были истинными мастерами своего дела, и все вырастало в большой неразберихе... [Когда FORTRAN уже начал распространяться] мы столкнулись с тем фактом, что не все из 25 000 его инструкций правильно работают и что возникают проблемы, которые можно выявить только после длительного использования».

К середине 60-х гг. (к этому времени и относится приведенное выше высказывание), после появления языков FORTRAN, COBOL, LISP и ALGOL, Бэкус уже понимал, что программирование изменилось. Машины становились дешевле, затраты на программирование, наоборот, росли, появилась реальная необходимость переносить программы с одной машины на другую, а поддержка конечного продукта требовала значительных компьютерных ресурсов. В связи с этим поменялись требования, которым должен был удовлетворять создаваемый язык программирования. Вместо обеспечения эффективной работы скомпилированной программы на дорогом компьютере перед языками высокого уровня возникла другая задача — упростить создание корректных программ для решения задач в конкретных областях.

Технология создания компиляторов была сформирована в 60-е и 70-е гг. (см. главу 3), и развитие языковых технологий сконцентрировалось на решении специфичных задач в конкретных областях. В научном программировании в основном использовался FORTRAN, деловые приложения писались на языке COBOL, в военной сфере применялся JOVIAL, программы искусственного интеллекта писались на LISP, встроенные военные приложения использовали Ada.

Как и естественные языки, языки программирования развиваются и в конце концов выходят из употребления, умирают. Так, язык ALGOL использовался в 60-е, затем его сменил Pascal, который, в свою очередь, вытесняется языками C++ и Java. В области деловых приложений уменьшается роль языка COBOL, его также заменяет C++. В 60-е гг. активно применялись языки APL, PL/1 и SNOBOL4, а в 70-е — Pascal, в настоящее же время они практически исчезли.

Те старые языки, которые применяются и в настоящее время, постоянно пересматриваются, чтобы соответствовать изменениям в других областях компьютерных технологий. Более новые языки, такие как C++, Java и ML, созданы на основе опыта, накопленного в процессе использования этих и сотен других более старых языков. На принципы конструирования новых языков влияют следующие факторы:

1. *Возможности компьютеров.* Компьютеры эволюционировали от огромных, медленных и дорогих ламповых машин 50-х гг. до современных суперкомпьютеров и микрокомпьютеров. В то же время между аппаратной частью компьютера и языком программирования появилось промежуточное звено, представляющее собой программное обеспечение операционных систем. Эти факторы оказали влияние как на структуру языков, так и на стоимость использования тех или иных языковых возможностей.

2. *Области применения.* В 50-е гг. компьютеры использовались лишь в военной отрасли, науке, деловом мире и промышленности, где высокая стоимость

была обоснованной. В настоящее же время их применение распространилось на область компьютерных игр, программ для персональных компьютеров, Интернета и вообще на приложения во всех областях человеческой деятельности. Требования, специфические для этих новых областей применения, влияют как на конструирование новых языков, так и на пересмотр и расширения старых языков.

3. *Методы программирования.* Структурное строение языка отражает изменяющееся с течением времени наше представление о том, *что* является хорошим методом написания большой и сложной программы, а также отражает изменяющуюся со временем среду, в которой осуществляется программирование.
4. *Методы реализации.* Усовершенствование методов реализации отражается на выборе тех новых свойств, которые добавляются во вновь разрабатываемые языки.
5. *Теоретические исследования.* Исследование концептуальных основ разработки и реализации языка с помощью формальных математических методов углубляет понимание сильных и слабых сторон конкретного языка, что отражается на добавлении тех или иных свойств при создании новых языков.
6. *Стандартизация.* Необходимость в стандартных языках, которые могут быть легко реализованы в различных компьютерных системах (что позволяет переносить программы с одного компьютера на другой), сильно влияет на эволюцию принципов разработки языков программирования.

В табл. 1.2 кратко описаны факторы, оказавшие наиболее важное влияние на развитие языков программирования во второй половине XX столетия.

Таблица 1.2. Факторы, повлиявшие на развитие языков программирования

Годы	Факторы и новые технологии
1951–1955	**Аппаратная часть:** компьютеры на электронных лампах; память с ртутной линией задержки. **Методы:** языки ассемблера; основные концепции; подпрограммы; структуры данных. **Языки:** экспериментальное использование компиляторов выражений
1956–1960	**Аппаратная часть:** запоминающие устройства на магнитных лентах; память на сердечниках; схемы на транзисторах. **Методы:** ранние технологии компилирования; НФБ-грамматики; оптимизация кода; интерпретаторы; методы динамического распределения памяти и обработка списков. **Языки:** FORTRAN, ALGOL 58, ALGOL 60, LISP
1961–1965	**Аппаратная часть:** семейства совместимых архитектур, запоминающие устройства на магнитных дисках. **Методы:** мультипрограммные операционные системы; синтаксические компиляторы. **Языки:** COBOL, ALGOL 60 (новая версия), SNOBOL, JOVIAL
1966–1970	**Аппаратная часть:** увеличение размера и быстродействия при уменьшении стоимости; микропрограммирование; интегральные схемы. **Методы:** системы с разделением времени; оптимизирующие компиляторы; системы написания трансляторов. **Языки:** APL, FORTRAN 66, COBOL 65, ALGOL 68, SNOBOL4, BASIC, PL/1, SIMULA 67, ALGOL-W

Годы	Факторы и новые технологии
1971–1975	**Аппаратная часть:** мини-компьютеры; запоминающие устройства небольшой емкости; полупроводниковая память. **Методы:** верификация программ; структурное программирование; технологии программирования. **Языки:** Pascal, COBOL 74, PL/1 (стандарт), C, Scheme, Prolog
1976–1980	**Аппаратная часть:** микрокомпьютеры; запоминающие устройства большой емкости; распределенные вычисления. **Методы:** абстракция данных; формальная семантика; технологии программирования: параллельная, встроенная и в режиме реального времени. **Языки:** Smalltalk, Ada, FORTRAN 77, ML
1981–1985	**Аппаратная часть:** персональные компьютеры; рабочие станции; видеоигры; локальные вычислительные сети; ARPANET. **Методы:** объектно-ориентированное программирование; интерактивные среды разработки; синтаксические редакторы. **Языки:** Turbo Pascal, Smalltalk-80, Prolog, Ada 83, Postscript
1986–1990	**Аппаратная часть:** эра микрокомпьютеров, автоматизированное рабочее место (АРМ) проектировщика, архитектуры RISC, Интернет. **Методы:** клиент-серверные вычисления. **Языки:** FORTRAN 90, C++, SML (Standart ML)
1991–1995	**Аппаратная часть:** очень быстрые и недорогие рабочие станции и микрокомпьютеры; архитектура с массовым параллелизмом; звук, видео, факс, мультимедиа. **Методы:** открытые системы, среды разработки. **Языки:** Ada 95, языки создания процессов (TCL, Perl), HTML
1996–2000	**Аппаратная часть:** компьютеры — дешевые приспособления; персональный электронный помощник; Всемирная паутина WWW; домашние кабельные сети; большой объем дисковой памяти (гигабайты). **Методы:** электронная коммерция. **Языки:** Java, Javascript, XML

1.3.1. Какой язык следует считать хорошим?

Механизмы разработки языка высокого уровня должны постоянно совершенствоваться. Каждый язык, описанный в данной книге, имеет свои недостатки, но тем не менее все они относительно удачны по сравнению с сотнями других языков, которые были разработаны, реализованы, использовались какое-то время, а потом были преданы забвению.

Некоторые причины успеха или неуспеха языка могут быть внешними по отношению к самому языку. Так, использование языков COBOL или Ada в Соединенных Штатах для разработки приложений в некоторых предметных областях было регламентировано указом правительства. Аналогично часть успеха языка FORTRAN можно отнести к его большой поддержке различными производителями вычислительной техники, которые тратили много усилий на его изящные реализации и подробные описания. Часть успеха SNOBOL4 в 70-е гг. можно приписать превосходному описанию этого языка, сделанному Грисволдом [46]. Широкое распространение таких языков, как LISP и Pascal объясняется как их использованием в качестве объектов теоретического изучения студентами, специализировав-

шимися в области разработки языков программирования, так и реальной практической значимостью этих языков.

Свойства хорошего языка

Несмотря на большое влияние некоторых из перечисленных внешних причин, в конце концов, именно программисты иногда, может быть, косвенно, решают, каким языкам жить, а каким нет. Существует множество причин, по которым программисты предпочитают тот или иной язык. Рассмотрим некоторые из них.

1. *Ясность, простота и единообразие понятий языка.* Язык программирования обеспечивает как систему понятий для обдумывания алгоритмов, так и средства выражения этих алгоритмов. Язык должен стать помощником программиста задолго до стадии реального кодирования. Он должен предоставить ясный, простой и единообразный набор понятий, которые могут быть использованы в качестве базисных элементов при разработке алгоритма. С этой целью желательно иметь минимальное количество различных понятий с как можно более простыми и систематизированными правилами их комбинирования. Это свойство мы называем *концептуальной целостностью.* Синтаксис языка влияет на удобство и простоту написания и тестирования программы, а в дальнейшем способствует ее пониманию и модификации. Центральным моментом здесь является удобочитаемость программы. Слишком лаконичный синтаксис может оказаться удобным при написании программ (особенно для опытного программиста), однако, когда такую программу нужно модифицировать, в ней оказывается нелегко разобраться. Программы на APL обычно настолько непонятны, что даже их разработчики спустя несколько месяцев после завершения работы затрудняются их расшифровать. Многие языки содержат такие синтаксические конструкции, которые сами подталкивают к неправильному восприятию программ, поскольку два почти одинаковых оператора на самом деле имеют кардинально различные значения. Например, появление пробела в операторе языка SNOBOL4 может полностью изменить его смысл. Хороший язык характеризуется тем, что конструкции, *обозначающие* различные понятия, должны и выглядеть совершенно по-разному, то есть семантические отличия должны отражаться в синтаксисе.
2. *Ортогональность.* Термин «*ортогональность*» означает, что любые возможные комбинации различных языковых конструкций будут осмысленными. Например, предположим, что язык позволяет задать выражение, которое вычисляет численное значение, а также задать условный оператор, в котором вычисляется выражение, с единственной целью — получить булево значение истина или ложь. Эти две языковые конструкции (выражение и условный оператор) ортогональны, если любое выражение можно использовать (и вычислять) внутри условного оператора.

   Когда конструкции языка ортогональны, язык легче выучить и на нем легче писать программы, поскольку в нем меньше исключений и специальных случаев, требующих запоминания. Отрицательной стороной ортогональности является то, что программа никогда не будет выдавать ошибки при компи-

ляции, даже если она содержит комбинацию возможностей, которые логически не согласованы или крайне неэффективны при выполнении.

3. *Естественность для приложений.* Язык должен иметь такой синтаксис, который при правильном использовании позволяет отражать в самой структуре программы лежащие в основе реализуемого ею алгоритма логические структуры. В идеале должна существовать возможность прямого перевода эскиза такой программы в подходящие программные операторы, отражающие структуру алгоритма. Последовательный, параллельный, логический и другие алгоритмы имеют различные естественные структуры, которые представлены в программах, написанных на языках, предназначенных для реализации этих алгоритмов. Язык должен предоставлять соответствующие решаемой задаче структуры данных, операции, структуры управления и естественный синтаксис. Одной из важнейших причин распространения того или иного языка является его естественность. Язык, соответствующий определенному классу приложений, может сильно облегчить создание отдельных программ в этой области. В качестве примера языков с очевидной направленностью на решение конкретных классов задач можно привести Prolog с уклоном в сторону дедукции и C++, предназначенный для объектно-ориентированных разработок.

4. *Поддержка абстракций.* Даже в наиболее естественном для данной предметной области языке программирования остается некоторый существенный пробел. Это пробел между абстрактными структурами данных и операциями, которые характеризуют решение задачи, и конкретными базовыми структурами данных и операциями, встроенными в язык. Например, язык C вполне может подойти для составления расписания занятий в университете, хотя в нем напрямую не существует естественных для этого приложения абстрактных структур данных, таких как *студент, курс лекций, преподаватель, аудитория,* и таких абстрактных операций, как *определить студента в группу* и *назначить аудиторию для группы.* Важной частью работы программиста является разработка конкретных абстракций для решения задачи и их последующая реализация с использованием базовых возможностей реального языка программирования. В идеале, язык должен позволять определять структуры данных, типы данных и операции и поддерживать их как самодостаточные абстракции. В этом случае программист сможет использовать их в других частях программы, зная только их абстрактные свойства и не вникая в их фактическую реализацию. Как Ada, так и C++ были разработаны именно по причине отсутствия этой возможности в более ранних языках, таких как Pascal и C соответственно.

5. *Удобство верификации программы.* Основным требованием является надежность программы, написанной на том или ином языке. Существует множество технологий для проверки правильности выполнения программой своих функций. Правильность программы можно доказать с помощью формальных методов верификации (см. раздел 4.2), проверкой без выполнения (путем чтения текста программы и исправления ошибок), также она может быть *протестирована* путем ее выполнения с тестовыми входными данными и

проверкой выходных результатов в соответствии с ее спецификацией и т. д. Для проверки больших программ обычно используется некая комбинация всех этих методов. Даже если язык обеспечивает на первый взгляд много возможностей для облегчения программирования, но проверка написанных на нем программ затруднительна, он менее надежен, чем язык, поддерживающий и упрощающий проверку программы. Основной фактор, влияющий на упрощение проверки программы, — простота семантики и синтаксических структур.

6. *Среда программирования.* Наличие в языке программирования технически развитых конструкций и структур (выражений, структур управления, типов данных и структур данных) — это только один аспект, влияющий на широту его использования. Наличие подходящей среды программирования может сделать технически слабый язык более легким в применении, нежели сильный язык при незначительной внешней поддержке. Можно составить длинный список разнообразных определяющих факторов, которым должна удовлетворять среда программирования, но возглавляет его, несомненно, требование наличия в ней надежной, эффективной и хорошо документированной реализации языка программирования. Специализированные текстовые редакторы и тестирующие пакеты, которые отражают специфику языка и работы с ним, могут сильно ускорить написание и тестирование программ. Средства для поддержки и модификации нескольких версий программы могут облегчить разработку больших программ. Из языков, описанных в данной книге, только Smalltalk был специально разработан под конкретную среду программирования, состоящую из окон, меню, возможностей ввода данных с помощью мыши и набора средств, позволяющих оперировать с программой, написанной на Smalltalk.
7. *Переносимость программ.* Одним из важных критериев для многих программных проектов является *переносимость* разработанных программ с компьютера, на котором они были написаны, на другие компьютерные системы. Удобным средством создания переносимых программ являются широкодоступные языки, определение которых не зависит от возможностей различных машин. Такие языки, как Ada, FORTRAN, C и Pascal имеют стандартные определения, позволяющие реализовывать переносимые приложения. Другие языки (например, ML) происходят от единственной централизованной реализации (single-source implementation), позволяя разработчику языка осуществлять некоторый контроль над свойствами, определяющими его переносимость.
8. *Стоимость использования.* Коварный критерий стоимости был оставлен напоследок. Стоимость использования, конечно, является существенным компонентом оценки языка программирования и складывается из нескольких составляющих:
   - *Стоимость выполнения программы.* На заре компьютерных вычислений стоимость связывалась в основном только с выполнением программы. Большое значение имели исследования по разработке оптимизирующих компиляторов, эффективного использования регистров и механизмов

эффективного выполнения программ. Хотя стоимость выполнения программы учитывается и в процессе разработки языка, но в первую очередь она важна для больших производственных программ, которые многократно выполняются. Однако на сегодняшний день для большинства приложений вопрос скорости выполнения не является первостепенным. Если речь идет об улучшении диагностики или упрощении контроля в процессе разработки и сопровождения программы, то при использовании настольных компьютеров, выполняющих много миллионов операций в секунду и большую часть времени находящихся в режиме ожидания, увеличение времени выполнения на 10 или 20 % является допустимым.

- *Стоимость трансляции программы.* Когда такие языки, как FORTRAN или C используются в процессе обучения, первостепенным может оказаться вопрос эффективной трансляции (компиляции), а не эффективного выполнения. Как правило, в процессе отладки студенческие программы многократно транслируются, а выполняются всего несколько раз. В этом случае важнее иметь быстрый и эффективный компилятор, а не компилятор, создающий эффективный код.
- *Стоимость создания, тестирования и использования программы.* Этот аспект стоимости может быть проиллюстрирован на примере языков Smalltalk и Perl. Для определенного класса задач решение может быть разработано, закодировано, протестировано, изменено и использовано с минимальными затратами времени и сил программиста. Smalltalk и Perl являются примерами эффективных в смысле стоимости языков программирования. И это объясняется тем, что в них минимизировано общее время и объем усилий, требующихся программисту на решение на компьютере какой-либо задачи, даже если время выполнения программы может быть больше, чем для других языков.
- *Стоимость сопровождения программы.* Многочисленные исследования показали, что самую большую часть стоимости программы, используемой в течение нескольких лет, составляет не стоимость начального создания, кодирования и тестирования программы, а *стоимость полного жизненного цикла* программы, куда входит стоимость как разработки, так и сопровождения программы. Поддержка включает в себя и исправление ошибок, выявленных уже после того, как программа отдана в эксплуатацию, и изменения, которые необходимо внести в программу в связи с обновлением аппаратной части или операционной системы, и усовершенствование и расширение возможностей программы для удовлетворения новых потребностей. Язык, который позволяет без особых проблем вносить многочисленные изменения и исправления в программу и создавать различные расширения (причем разными программистами и в течение многих лет), окажется в конечном счете более выгодным, чем любой другой.

Синтаксис и семантика

Синтаксис языка программирования определяет то, как выглядит программа на этом языке. Определить синтаксические правила — значит объяснить, как пишут-

ся операторы, объявления и другие языковые конструкции. *Семантика* языка программирования определяет смысловые значения различных синтаксических конструкций. Например, чтобы в языке С задать вектор V из десяти целочисленных элементов, необходимо написать следующее объявление:

```
int V[10]:
```

В языке же Pascal то же самое будет выглядеть по-другому:

```
V: array[0..9] of integer:
```

Как видно, их синтаксис сильно различается, хотя во время выполнения программы создаются одинаковые объекты данных. Чтобы понять смысл этих объявлений, необходимо знать семантику объявлений массивов в языках С и Pascal. То есть вы должны знать, что если поместить подобное объявление в начало подпрограммы, то при каждом ее вызове будет создаваться вектор, содержащий указанное число элементов, а по завершении программы он будет уничтожаться. Во время выполнения подпрограммы на вектор можно ссылаться по имени V. В обоих примерах элементами вектора V будут V_0, ..., V_9.

Однако, если вы объявляете объект V как список в языке LISP, необходимо знать, что размер его может быть произвольным и определяется непосредственно в момент создания. Более того, объект может быть создан в любой момент во время выполнения программы и на первый элемент можно ссылаться одним из следующих способов: `(car V)` или `(head V)`.

В руководствах по языкам программирования и в справочниках принято строить описание языка на основе его синтаксических конструкций. Как правило, сначала приводится синтаксис некоторой языковой конструкции, например конкретного оператора или объявления, а затем поясняется семантика этой конструкции, то есть описывается ее смысл. В главе 3 приведена система обозначений НФБ, которая является в настоящее время основной системой обозначений, используемой для описания синтаксиса языка программирования.

В этой книге мы придерживаемся другого способа описания; он основан на структурах, связанных с выполнением программы. Иногда такие структуры данных и операции непосредственно связаны с конкретными синтаксическими конструкциями, но чаще эта связь является более опосредованной. Например, выполняемая программа на языке С может использовать вектор V, где V имеет структуру, определенную приведенным выше объявлением. Однако скомпилированная программа может иметь другие структуры данных, такие как центральный стек, содержащий активационные записи подпрограмм, которые напрямую не связаны с синтаксисом программы.

Эти скрытые структуры так же важны для понимания языка, как и видимые структуры, напрямую соответствующие тому, что программист написал в программе. Именно поэтому в данной книге рассмотрение элементов языка организовано вокруг выполняемых структур, а не синтаксических элементов. Конкретный элемент, создаваемый во время выполнения программы, может не иметь синтаксического представления в программе, может быть представлен напрямую единственным синтаксическим элементом, может выражаться через несколько различных синтаксических элементов, которые собираются транслятором воедино для создания одного элемента виртуального компьютера.

1.3.2. Парадигмы языка

Когда собираются приверженцы различных языков, разговор обычно принимает форму политического митинга. Жаркая дискуссия об эффективности объявления массивов в C++ по сравнению с Java или о преимуществе интерпретации программ перед их компиляцией здесь обеспечена. Хотя, по правде говоря, способ объявления массивов и вопросы трансляции слабо связаны с различиями, которые существуют между этими языками. Всегда имеются незначительные синтаксические отличия, которые просто отражают индивидуальные склонности разработчиков языков и которые практически не влияют на программы, написанные на этих языках. Чтобы понять, как устроен язык, необходимо заглянуть глубже.

Существуют четыре основные вычислительные модели, которые описывают большинство сегодняшних методов программирования: императивная, аппликативная, основанная на системе правил и объектно-ориентированная. Далее мы приводим краткое описание каждой модели.

Императивные языки. *Императивные*, или *процедурные, языки* — это управляемые командами или операторно-ориентированные языки программирования. Основной концепцией является состояние машины — множество всех значений всех ячеек памяти компьютера. Программа состоит из последовательности операторов, выполнение каждого из которых влечет за собой изменение значения в одной или нескольких ячейках памяти, то есть переход машины в новое состояние. В целом синтаксис такого языка имеет вид:

```
оператор1:
оператор2:
...
```

Рис. 1.1, *а* отображает этот процесс. Представим, что память состоит из набора *шариков в коробочках*, а выполнение оператора (например, сложение двух переменных для получения значения третьей) может быть представлено как обращение к ячейке памяти (коробочке), комбинирование тем или иным способом значений (шариков) и сохранение результата (нового шарика) в новой ячейке. Разработка программы состоит в построении последовательных состояний машины, необходимых для достижения результата. Обычно при первом знакомстве с концепциями программирования люди сталкиваются именно с этой моделью, и многие широко распространенные языки поддерживают именно ее (например, C, C++, FORTRAN, ALGOL, PL/1, Pascal, Ada, Smalltalk и COBOL). Эта модель вытекает из особенностей аппаратной части стандартного компьютера, выполняющей инструкции (команды) последовательно. Неудивительно, что большинство традиционных языков следуют именно этой модели.

Аппликативные языки. Другим взглядом на вычисления, производимые с помощью языка программирования, является рассмотрение функции, которую выполняет программа, а не отслеживание изменяемых состояний машины во время выполнения программы, оператор за оператором. Мы можем достичь этого, сосредоточив внимание на ожидаемом результате, а не на имеющихся в нашем распоряжении данных. Другими словами, вместо того чтобы рассматривать последовательность состояний, через которые должна пройти машина для получения ответа, вопрос следует поставить по-другому: какую функцию необходимо применить к начальному состоянию машины (путем выбора начального набора переменных и ком-

бинирования их определенным образом), чтобы получить требуемый ответ? Языки, в которых акцентирован именно этот взгляд на вычисления, называются *аппликативными*, или *функциональными*.

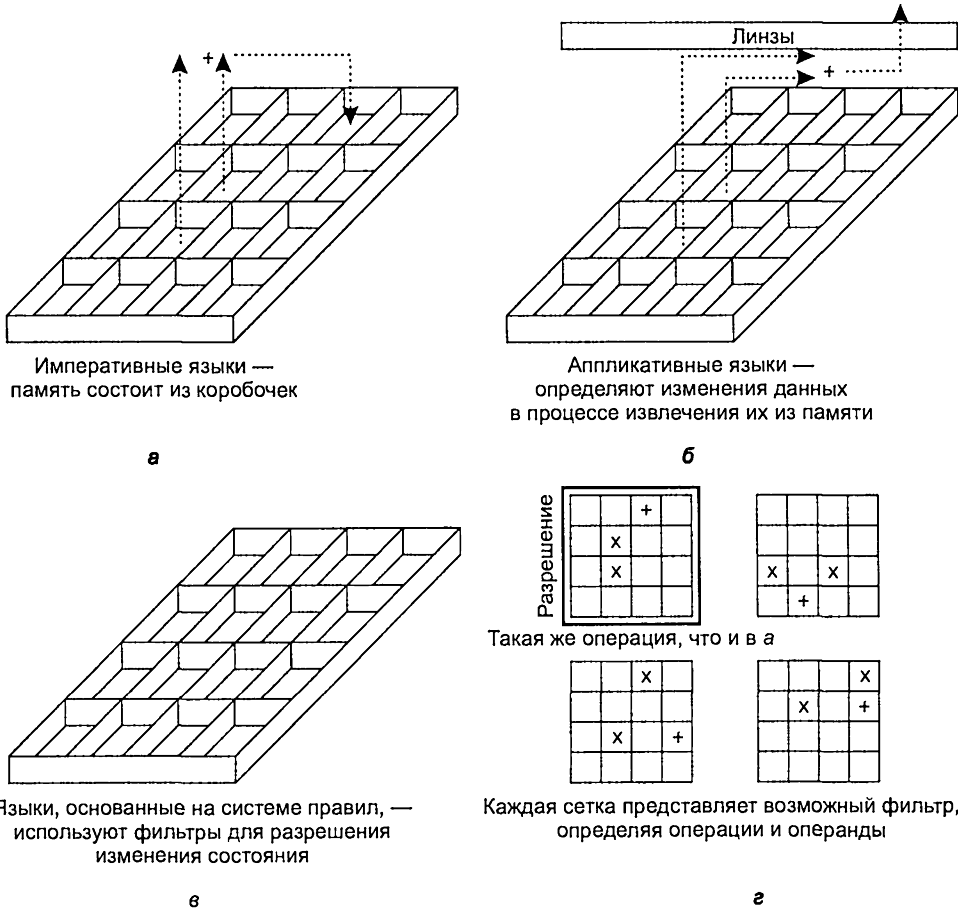


Рис. 1.1. Вычислительные модели языков программирования

Эта модель представлена на рис. 1.1, *б* с помощью линзы, которая получает начальные данные и путем воздействия на их представление в памяти (как при прохождении света через линзу) производит желаемый ответ. Разработка программ сводится к созданию из уже имеющихся простых функций более сложных функций, которые поочередно воздействуют на начальный набор данных до тех пор, пока последняя функция не выдаст требуемый результат. Вместо того чтобы рассматривать последовательные состояния машины, в данной модели мы концентрируем наше внимание на последовательных преобразованиях начальных данных с помощью функций. Как только у нас есть конечный ответ (последовательность функций), его можно применить к начальным данным и получить результат. Синтаксис такого языка, как правило, выглядит следующим образом:

$функция_n(...функция_2(функция_1(данные))...)$

Из описанных в данной книге языков функциональную модель поддерживают ML и LISP.

Языки, основанные на системе правил. Языки, основанные на системе правил, осуществляют проверку наличия необходимого разрешающего условия и в случае его обнаружения выполняют соответствующее действие. Наиболее распространенный язык, основанный на системе правил, — Prolog. Он называется также *языком логического программирования*, поскольку базовые разрешающие условия относятся к классу выражений логики предикатов (более подробно данная тема будет рассмотрена в разделе 8.4.2). На рис. 1.1, *в* языки, основанные на системе правил, схематически представлены с помощью набора фильтров, которые нужно применить к данным, находящимся в памяти. Выполнение программы на подобном языке похоже на выполнение программы, написанной на императивном языке, за исключением того, что операторы выполняются не в той последовательности, в которой они определены в программе. Разрешающие условия определяют порядок выполнения. Синтаксис таких языков выглядит следующим образом:

```
разрешающее условие1 → действие1
разрешающее условие2 → действие2
...
разрешающее условиеn → действиеn
```

(Иногда правила записываются в виде `действие if разрешающее условие`, в котором выполняемое действие записывается слева.)

Хотя Prolog — наиболее известный из языков такого класса, существует также множество других языков, использующих эту парадигму. Обыкновенное бизнес-приложение таблиц решений является формой языка, основанного на системе правил. В данном случае разрешающие условия для данных обусловливаются наличием или отсутствием допустимых значений в записях данных. Программирование обычно сводится к построению матрицы (таблицы) возможных условий и заданию соответствующих действий в случае выполнения этого условия (отсюда и название). Методы синтаксического разбора НФБ (описываемые в главе 3) и средства синтаксического разбора типа YACC (Yet Another Compiler Compiler) относятся к технологиям, основанным на системе правил, в которых формальный синтаксис программы рассматривается в качестве разрешающего условия.

Объектно-ориентированное программирование. Как будет показано в главе 7, значимость объектно-ориентированной модели постоянно возрастает. В этой модели строятся сложные объекты данных, а затем для операций над этими данными описывается ограниченный набор функций. Сложные объекты создаются как расширения более простых объектов и наследуют их свойства. Как мы покажем, на самом деле эта модель является попыткой объединить лучшие свойства других моделей. Благодаря возможности строить конкретные объекты данных объектно-ориентированная программа приобретает эффективность императивного языка. Построение классов функций, которые используют ограниченный набор объектов данных, дает нам гибкость и надежность, свойственные аппликативному языку.

Универсальность вычислительной модели

При описании конкретных вычислительных моделей мы специально старались использовать термин *«поддерживать»*, а не *«реализовывать»*. То, как программист

использует язык, зависит только от него самого. Императивный язык упрощает операторно-ориентированное программирование, управляющее внутренним машинным состоянием компьютера; но, в принципе, можно написать программы, которые будут выполняться последовательно и выполнять те же функции, на языках LISP или Prolog. Также на языке С относительно легко написать программу, состоящую только из вызовов функций и поэтому похожую на аппликативную. Об этом необходимо помнить, когда в дальнейшем будут обсуждаться различные возможности и свойства языков.

Исторически сложилось так, что императивные языки составили первый широко используемый класс языков и в настоящее время остаются доминирующими в программировании. Одним из наиболее интересных результатов исследований 70–80-х гг. оказался тот факт, что аппликативная методика обеспечивает эффективные способы верификации программ и доказательства их корректности. Это видно из блок-схем, представленных на рис. 1.2. На рис. 1.2, *а* изображена блок-схема, типичная для программ 60-х гг. В ней нет никакой явной структуры, и кажется, что передача управления происходит чуть ли не случайным образом. Сейчас такие программы обычно называют *программами-спагетти* (с большим числом нерациональных передач управления назад и вперед), поскольку маршруты управления напоминают тарелку со спагетти. В таких программах часто бывает трудно понять, каково состояние программы в каждый момент времени в процессе ее выполнения.

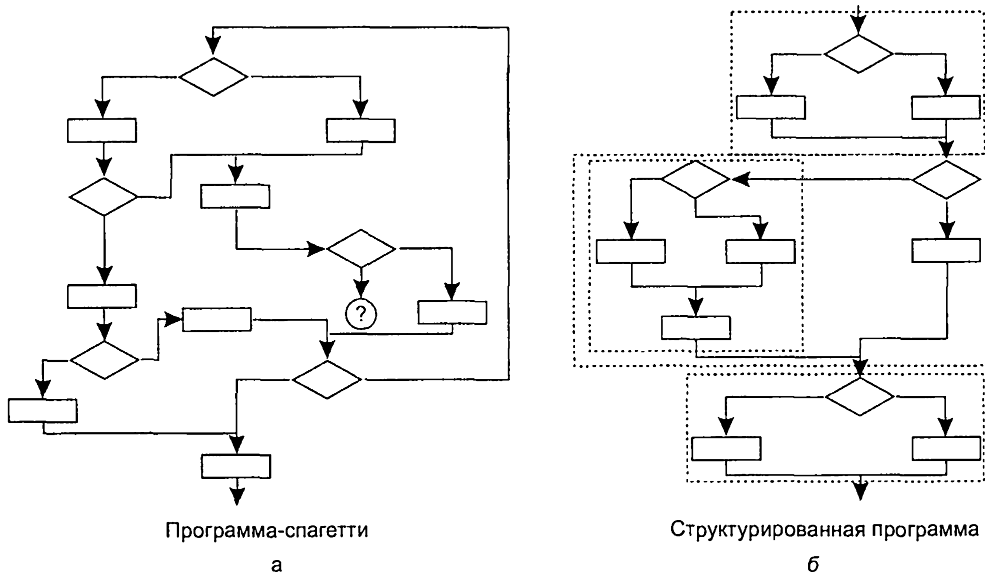


Рис. 1.2. Аппликативные методы в императивных языках

На рис. 1.2, *б* приведена более структурированная конструкция. Каждый сегмент данной блок-схемы можно заключить в пунктирный прямоугольник. Каждый из четырех прямоугольников на этой схеме имеет одну входную и одну выходную стрелки. Эту программу можно рассматривать как композицию четырех функций, и поведение программы можно определить как функцию, которая получает данное со-

стояние на входе выделенного пунктиром прямоугольника и преобразует его в результирующее состояние на выходе из него. Однако это всего-навсего форма аппликативной модели, описанной ранее. В дальнейшем, при описании простых программ в разделе 8.3.3, мы обсудим этот момент подробнее. В основе большинства исследований по проверке программ с использованием формальных доказательств правильности программы лежит применение аппликативных методов к императивным программам. Как будет показано в главе 4, некоторые из предикатных и алгебраических методов основаны на аппликативной модели. Также мы увидим, что объектно-ориентированное программирование является альтернативной возможностью применения аппликативной модели к императивным программам.

1.3.3. Стандартизация языка

Что описывает язык программирования? Рассмотрим следующий код, написанный на языке С:

```
int i; i=(1&&2)+3;
```

Действительно ли это правильный синтаксис С? Каково значение i? Как бы вы ответили на эти вопросы?[1] Как правило, чтобы ответить на подобные вопросы, применяют три подхода:

1. Прочитать в справочном руководстве по языку определение соответствующих операций и выяснить, что означает написанный код.
2. Написать программу и посмотреть, что она вычислит.
3. Прочитать определение операций в описании стандарта языка.

Второй способ, наверное, является самым распространенным, поскольку сесть и написать программу из двух-трех строчек и посмотреть, что получится, не составляет труда. Вообще-то концепция языка программирования тесно связана с конкретной его реализацией, используемой на локальном компьютере. Если вы предпочитаете более научный подход, можно обратиться к справочному руководству по языку, которое обычно издается поставщиком конкретного компилятора С. Поскольку немногие имеют доступ к описанию стандарта языка, третий способ применяется сравнительно редко.

Первый и второй способы подразумевают, что концепция языка программирования связана с его конкретной реализацией. Но является ли эта реализация правильной? А что, если понадобится перенести программу, написанную на языке С и состоящую из 50 000 строчек, на другой компьютер, где установлен компилятор С от другого поставщика? По-прежнему ли программа будет правильно скомпилирована и при выполнении выдаст тот же результат? Если нет, то почему? Зачастую идеология языка включает в себя некоторые неочевидные детали, интерпретация которых может различаться в различных реализациях, что порождает несколько разное поведение программы при ее выполнении на различных компьютерах.

Однако некоторые поставщики программного обеспечения могут решить, что новая возможность, добавленная в язык при его реализации, пойдет ему на пользу.

[1] Заметим (для любопытных), что ответы на первый и второй вопросы соответственно «да» и «4».

Допустимо ли такое? Например, если добавить в язык C новый способ объявления динамических массивов, можно ли язык с подобным расширением по-прежнему называть языком C? Если это случится, то программу, использующую эту новую возможность и компилирующуюся на локальном компиляторе, в котором подобная возможность реализована, невозможно будет скомпилировать в другой системе, использующей другой компилятор языка.

Для разрешения подобных проблем многие языки имеют стандартные определения. Все реализации должны придерживаться этого стандарта. Стандарты обычно бывают двух видов.

1. *Частный стандарт.* Сюда входят определения, сделанные той компанией, которая разработала язык и имеет на него авторские права. В большинстве случаев для популярных и широко используемых языков такие стандарты не работают, поскольку в этих случаях часто появляются новые реализации, усовершенствованные и несовместимые.
2. *Согласительный стандарт.* К нему относятся созданные специальными организациями документы, основанные на соглашении всех заинтересованных участников. Согласительный стандарт, или просто *стандарт*, является основным способом обеспечения единообразия различных реализаций языка.

В каждой стране, как правило, есть одна или несколько организаций, наделенных правом разработки стандартов. В Соединенных Штатах это Американский национальный институт стандартов (ANSI — American National Standards Institute). Стандарты языков программирования могут разрабатываться комитетом X3 Ассоциации производителей делового компьютерного оборудования (CBEMA — Computer Business Equipment Manufactures Association), а также Институтом инженеров по электротехнике и электронике (IEEE — Institute of Electrical and Electronic Engineers). В Великобритании такими полномочиями наделен Британский институт стандартов (BSI — British Standards Institute). Международные стандарты создаются Организацией международных стандартов (ISO — International Standards Organization), штаб-квартира которой находится в Женеве (Швейцария).

В Соединенных Штатах следование стандартам — дело добровольное. Национальный институт стандартов и технологий (NIST — National Institute of Standards and Technology), являющийся правительственным органом, создает федеральные стандарты. Выполнение этих стандартов является обязательным только для поставщиков программного обеспечения, продающих продукцию федеральным органам. Частные же компании могут не следовать этим стандартам. Однако федеральные стандарты всегда принимаются в качестве общих, а NIST, ANSI, IEEE и ISO обычно разрабатывают стандарты совместно.

Процесс разработки стандартов во всех подобных организациях одинаков. В некоторый момент специальная группа решает, что язык необходимо стандартизовать. Затем совет по стандартам нанимает группу добровольцев для разработки стандарта. Когда эта рабочая группа вырабатывает свой вариант стандарта, он переходит на голосование в группу заинтересованных лиц. Расхождения, выявленные при голосовании, прорабатываются, и затем выпускается стандарт языка.

Хотя теоретически все выглядит хорошо, на самом деле создание стандартов является отчасти технической, а отчасти и политической проблемой. Например, поставщики компиляторов имеют определенный финансовый интерес в создании определенных стандартов. Естественно, они хотят, чтобы стандарт был как можно ближе к их компилятору, чтобы не пришлось вносить изменения в свою реализацию. Такие изменения не только дорого стоят сами по себе, они приводят еще и к тому, что программы пользователей, работающих с измененным компилятором, перестают отвечать стандарту. Это, разумеется, создает неудобства потребителям.

Поэтому, как и отмечалось выше, создание стандартов — процесс, основанный на компромиссах. Не каждый извлекает из него выгоду, но все же имеется надежда, что получившийся в результате язык будет приемлемым для всех. Рассмотрим следующий простой пример. В процессе обсуждения стандарта FORTRAN 77 все стороны пришли к соглашению, что поскольку большинство реализаций FORTRAN уже имели строки и подстроки, желательно стандартизовать такую возможность. Однако существовало несколько реализаций подстрок. Для пояснения рассмотрим строку `M = "abcdefg"`. Согласно одной реализации, подстрока "bcde" обозначалась `M[2:5]`, то есть строка, состоящая из символов строки `M` со второго по пятый. В другой реализации та же подстрока обозначалась `M[2:4]`, то есть строка, начинающаяся с позиции 2 и имеющая 4 символа. Если же считать символы справа налево, ту же подстроку можно было записать и как `M[3:6]`. Поскольку невозможно было достигнуть никакого соглашения по этому вопросу, было решено просто оставить его за рамками стандарта. Хотя этот стандарт не отвечает большинству высказанных в данной главе требований, принятое решение оказалось вполне подходящим. По этой причине стандарты являются полезными документами, но на определение языка могут оказать влияние некоторые сиюминутные политические соображения.

Чтобы использовать стандарт эффективно, следует рассмотреть три аспекта стандартизации.

1. *Своевременность*. Когда нужно стандартизовать язык?
2. *Соответствие*. Как определить, что программа соответствует стандарту, а компилятор компилирует согласно стандарту?
3. *Устаревание*. Что обусловливает устаревание стандарта и как его следует модифицировать?

Рассмотрим все эти вопросы по очереди.

Своевременность. Одним из важных вопросов является определение момента, *когда* следует стандартизовать язык. Язык FORTRAN впервые был стандартизован в 1966 г., когда уже существовало несколько несовместимых его версий. Это привело к некоторым проблемам, поскольку каждая реализация отличалась от других. Другой крайностью является язык Ada, который впервые был стандартизован в 1983 г., еще до того, как возникла первая реализация. Когда создавался этот стандарт, еще никто не мог сказать, будет ли вообще этот язык работать. Первые эффективные компиляторы Ada появились только в 1987 г., а некоторые недостатки этого языка были обнаружены только при создании этих первых реализаций. Таким образом, следует создавать стандарт языка не слишком рано, чтобы

накопилось достаточно опыта в его применении, но и не слишком поздно, чтобы не поощрять создание несовместимых реализаций.

Из языков, описанных в данной книге, FORTRAN был стандартизован слишком поздно, когда было уже много несовместимых реализаций, а Ada, наоборот, — слишком рано, когда еще не было ни одной. Вовремя появились стандарты языков С и Pascal, когда они уже начали распространяться, но еще не было большого количества несовместимых версий.

Соответствие. Если существует стандарт языка, то, естественно, возникает вопрос о *соответствии* этому стандарту. Программа считается *соответствующей стандарту*, если она использует только те возможности, которые определены данным стандартом. Компилятор является *соответствующим стандарту*, когда после компиляции соответствующей стандарту программы при ее выполнении получается правильный результат.

Следует отметить, что здесь ничего не говорится о расширениях стандарта. Если в компилятор добавлены дополнительные возможности, то любая программа, использующая эти возможности, не является соответствующей стандарту и стандарт ничего не говорит о том, какими должны быть результаты выполнения таких программ. Стандарт обычно относится только к соответствующим стандарту программам. Вследствие этого большинство компиляторов всегда имеют возможности, которые не описаны стандартом. Поэтому следует проявлять осторожность при использовании локальной реализации как окончательной инстанции при выяснении смысла какой-либо языковой конструкции. (В качестве примера можно рассмотреть программу anomaly на языке Pascal, приведенную в листинге 9.1.)

Устаревание стандартов. Накопленные знания и опыт программирования подсказывают, что развитие новых компьютерных архитектур требует включения в языки новых возможностей. Спустя несколько лет после утверждения стандарта языка, он может выглядеть весьма причудливо. Так устарел исходный стандарт FORTRAN 66, поскольку в нем не предусмотрены многие возможности современных языков, такие как типы, вложенные управляющие структуры, инкапсуляция, блочная структура и многое другое из арсенала современных языков программирования.

В связи с этим процесс стандартизации стал предусматривать возможность обновления. Каждый стандарт должен пересматриваться раз в пять лет и либо обновляться, либо совсем отменяться. Правда, всегда находится какая-либо причина для нарушения пятилетнего срока, но тем не менее такой процесс по большей части достаточно эффективен. Первый стандарт языка FORTRAN был принят в 1966 г., затем он был пересмотрен в 1978 г. (хотя срок завершения разработки стандарта в 1977 г. и был перенесен на несколько месяцев, язык все же называется FORTRAN 77), а затем в 1990 г. стандарт был снова пересмотрен. Язык Ada был стандартизован в 1983 г., а пересмотрен в 1995 г.

Один из вопросов, связанных с обновлением стандарта, заключается в следующем: что делать с уже существующим набором написанных в соответствии со старым стандартом программ? Компании затратили большие средства на создание программного обеспечения, а переписывать все существующие программы под новую версию языка — весьма дорогостоящее мероприятие. Поэтому в большин-

стве стандартов требуется предусматривать обратную совместимость, то есть новый стандарт должен включать в себя более старые версии языка.

Однако при этом возникают некоторые проблемы. Например, язык может стать слишком громоздким, если в нем накапливать все устаревшие конструкции. Более того, некоторые из таких конструкций могут препятствовать разработке хорошей программы. Оператор EQUIVALENCE языка FORTRAN может служить примером такой устаревшей конструкции. Пусть A вещественное число, а I — целое, тогда последовательность операторов

```
EQUIVALENCE (A.I)
A=A+1
I=I+1
```

располагает переменные A и I в одной и той же ячейке памяти. Первая процедура присваивания (с переменной A) обращается к этой ячейке как к содержащей вещественное число и добавляет к нему единицу. Вторая процедура присваивания (с переменной I) обращается к той же ячейке как к содержащей уже целое число и также добавляет к нему единицу. Поскольку представления целых и вещественных чисел в большинстве компьютеров различны, то результат может быть совершенно непредсказуемым. Поэтому сохранение этой конструкции языка нежелательно.

В связи с вышесказанным недавно сформировались такие понятия, как *устаревшая* и *не рекомендуемая* возможности. Возможность называется *устаревшей*, если она является кандидатом на исключение из языка уже в следующей версии стандарта. Такое понятие предупреждает пользователей, что пока еще эта возможность доступна, но в ближайшие 5–10 лет она будет изъята из стандарта языка. Таким образом, пользователи предупреждаются заблаговременно и имеют время переписать все коды, содержащие эту конструкцию. *Не рекомендуемая* возможность в очередном стандарте может стать устаревшей. Таким образом, она может быть исключена из языка после двух его последующих пересмотров. В этом случае предупредительный период составляет 10–20 лет. В новых программах такие возможности уже не следует использовать.

Поскольку расширения языка допускаются и в согласующихся со стандартом компиляторах (до тех пор, пока они правильно компилируют согласующиеся со стандартом программы), многие компиляторы имеют дополнения, которые с точки зрения поставщика программного обеспечения являются полезными и способствуют распространению программного продукта. Благодаря такому подходу появляются нововведения и развивается язык. Конечно, в академической среде большинство программистов не связывают себя никакими стандартами, а разрабатывают свои собственные продукты, в которых языки модифицированы и расширены подходящим для них образом. Это создает плодородную почву для испытания новых языковых концепций, а некоторые наиболее удачные идеи попадают в коммерческие языки и компиляторы.

1.3.4. Интернационализация

В связи с глобализацией торговли и появлением Всемирной паутины WWW программирование становится все более и более глобальной деятельностью, и поэто-

му важно, чтобы языки программирования могли легко использоваться в разных странах. Соответственно появляется необходимость, чтобы компьютеры могли «изъясняться» на разных языках. Например, для представления различных символов обычно недостаточно 8-битного байта, в котором можно закодировать лишь 256 различных символов. При обсуждении подобных вопросов обычно пользуются термином *интернационализация* (*internationalization*)[1].

Зачастую на способ хранения и обработки данных влияют местные соглашения. Такие моменты, как коды символов, схема упорядочения, формат даты и времени и другие местные стандарты влияют на вид входных и выходных данных. Некоторые из соответствующих факторов перечислены в работе Мартина [79].

- **Схемы упорядочения.** По какой схеме должно происходить упорядочение символов?
  - *Сортировка.* Позиции нелатинских символов, таких как Å, Ø, β, ö и других, не имеют единого определения и могут быть разными в разных странах.
  - *Регистр.* Некоторые языки, такие как японский, арабский, иврит и тайский, не имеют понятия о верхнем и нижнем регистрах.
  - *Направленность письма.* На многих языках читают слева направо, но бывает и по-другому (например, справа налево или сверху вниз).
- **Формат даты.** Формат даты зависит от страны. Так, в Америке принято писать 11/26/02 (26 ноября 2002 г.), в Англии же наоборот — 26/11/02. Во Франции и России пишут 26.11.02, а в Италии — 26-XI-02 и т. д.
- **Формат времени.** Формат времени также зависит от страны. В Америке пишут 5:40 p. m., а в Японии и России — 17:40, в Германии это же время выглядит как 17.40, а во Франции — 17h40.
- **Часовые пояса.** Хотя существует стандартное правило изменения местного времени на один час на каждые 15° долготы, на самом деле это скорее общий принцип, нежели реальность. Обычно часовые пояса различаются на целое количество часов, но некоторые могут различаться на 15 минут или на полчаса. Сдвиги времени (такие как летнее время в Америке и Европе) во всем мире осуществляются неодинаково. Произвести пересчет местного времени во всемирное стандартное время иногда бывает непросто. В южном полушарии переход на летнее время осуществляется противоположно тому, как это происходит в северном полушарии.
- **Идеографические системы.** Письменность некоторых языков основана на большом количестве иероглифов, а не на алфавите, состоящем из небольшого набора символов (например, в японском, китайском и корейском языках). Обычно для представления текста на таких языках требуется 16 бит.
- **Денежные единицы.** Представление денежных единиц также зависит от страны (используются, например, такие символы, как $, £, ¥).

[1] Иногда используется термин «проблема I18N», поскольку само слово «internationalization» слишком длинное (20 символов), и, кроме того, название I18N позволяет избежать споров о том, как писать слово «internationalization»: через британское «s» или американское «z».

1.4. Среда программирования

Понятие среды программирования, в которой программа создается и тестируется, наверняка знакомо большинству читателей этой книги. Однако, как правило, среда программирования меньше влияет на строение языка программирования, нежели операционная среда, использующаяся для запуска программ. В первую очередь среда программирования состоит из набора средств поддержки и активизирующих их команд. Каждое из средств поддержки также является программой и может быть использовано программистом как вспомогательное средство на одной или нескольких стадиях создания программы. Обычно в средства поддержки входят редакторы, отладчики, верификаторы, генераторы тестовых данных и программы печати.

1.4.1. Влияние на разработку языка

Среда программирования влияет на разработку языка преимущественно в двух основных областях: на возможности языка, облегчающие раздельную компиляцию и сборку программы из различных компонентов, и на возможности, облегчающие тестирование и отладку программы.

Раздельная компиляция. При разработке любой большой программы обычно желательно по отдельности разработать, закодировать и протестировать все компоненты программы, созданные отдельными программистами и группами разработчиков, прежде чем собирать ее в единое целое из этих отдельных блоков. При таком подходе требуется, чтобы структура языка допускала раздельное компилирование и выполнение подпрограмм и других компонентов программы в отсутствие остальных ее частей и их последующее объединение без каких-либо дополнительных изменений.

Раздельная компиляция может быть затруднена по той причине, что при компиляции отдельной подпрограммы ей может потребоваться следующая информация о других подпрограммах или о совместно используемых объектах данных.

1. Описание количества, порядка и типа параметров любой вызываемой подпрограммы позволяет компилятору проверить правильность вызова внешней подпрограммы. Также может потребоваться информация о том языке, на котором была написана внешняя подпрограмма. Зная это, компилятор может составить соответствующую вызывающую последовательность инструкций для преобразования данных и управляющей информации, необходимой внешней подпрограмме во время выполнения, именно в той форме, которая требуется для данной подпрограммы.
2. Объявление типа данных для любой используемой переменной позволяет компилятору определить способ представления каждой внешней переменной в памяти. Тогда ссылка на эту переменную может быть скомпилирована с использованием соответствующей формула доступа (например, правильный сдвиг внутри общего блока).
3. Определение тех внешних типов данных, которые используются для описания локальных переменных внутри подпрограммы. Это позволит компиля-

тору выделить для них память и вычислить формулы доступа для таких локальных переменных.

Чтобы предоставить компилятору информацию о раздельно скомпилированных подпрограммах, совместно используемых объектах данных и определении типов, язык может:

1) либо требовать, чтобы эта информация *объявлялась еще раз* внутри подпрограммы (как в языке FORTRAN);
2) либо предписывать конкретный *порядок компиляции*, то есть перед компиляцией каждой подпрограммы должна производиться компиляция определений всех вызываемых подпрограмм и совместно используемых данных (как в языке Ada и до некоторой степени в Pascal);
3) либо требовать при компиляции наличия библиотек, содержащих все соответствующие определения, так чтобы компилятор мог получать их по мере необходимости (как в языках Java и C++).

Термин *«раздельная компиляция»* обычно используется для первого варианта. В этом случае каждая подпрограмма может быть скомпилирована без какой-либо внешней информации и является вполне самодостаточной. Недостатком раздельной компиляции является то, что обычно нет возможности проверить согласованность информации о внешних подпрограммах и данных, которые еще раз объявляются внутри подпрограммы. Если описания внутри подпрограммы не соответствуют действительным структурам внешних данных и подпрограмм, то на этапе сборки программы возникает трудноуловимая ошибка, которую не обнаружить при раздельной компиляции различных частей программы.

Варианты 2 и 3 требуют наличия в языке возможности того, чтобы спецификации подпрограмм, типов данных и общего окружения были предоставлены или помещены в библиотеку еще до компиляции подпрограммы. Для этих целей желательно, чтобы язык позволял определять подпрограммы с единственным обязательным заданием раздела *спецификаций,* а *тело* подпрограммы (локальные переменные и операторы) могло бы быть опущено. При этом следует предусмотреть возможность в последующем отдельной компиляции тела подпрограммы. В языке Ada, например, любая подпрограмма, задача или пакет разбиты на две части: спецификация и тело, которые могут быть скомпилированы по отдельности или помещены в библиотеку, если это требуется для компиляции других подпрограмм. Вызов подпрограммы, которая еще не была скомпилирована, называется *заглушкой.* Подпрограмма, содержащая заглушки, может быть выполнена. В тот момент, когда очередь доходит до заглушки, вместо вызова подпрограммы происходит вывод на печать системного диагностического сообщения (или осуществляется какое-либо другое действие). Таким образом, раздельно скомпилированную программу можно запускать на тестовое выполнение, даже если код некоторых вызываемых из нее подпрограмм еще недоступен.

Другим аспектом раздельной компиляции, влияющим на строение языка, является наличие совместно используемых переменных. Если различные группы разработчиков пишут отдельные части большой программы, обычно бывает трудно гарантировать, что все имена переменных, используемые каждой группой для подпрограмм и общего окружения, будут различными, то есть не произойдет слу-

чайного совпадения имен. Обычным делом при сборке программы в единое целое является обнаружение одинаковых имен в различных подпрограммах или других компонентах программы. Как правило, это приводит к кропотливому и трудоемкому исправлению уже оттестированного кода. Для избежания этой проблемы в языках применяется три метода.

1. Каждое совместно используемое имя (например, в операторе extern языка C) должно быть уникальным. Обеспечение выполнения этого требования входит в обязанности программиста. В начале работы должно быть составлено соглашение об использовании имен, так чтобы каждой группе выделялся свой набор имен для использования в подпрограммах (например, все имена, используемые некоторой группой, должны начинаться с QQ). Так, все имена, используемые в стандартных, включаемых директивой #include, файлах C, начинаются с символа _, поэтому программистам не рекомендуется использовать имена, начинающиеся с символа подчеркивания.
2. Для того чтобы скрыть имена, в языках программирования часто используются *правила определения области видимости*. Если в некоторой подпрограмме содержится еще одна подпрограмма, то другим раздельно компилируемым программам будет известно только имя самой внешней подпрограммы. Этот механизм используется в таких языках, как Pascal, C и Ada. Более подробно блочная структура и область видимости имен рассмотрены в разделе 9.2.2.
3. Имена могут становиться известными путем явного добавления их определений из внешней библиотеки. В объектно-ориентированных языках этот способ является основным механизмом реализации *наследования*. Путем добавления описания внешних классов в подпрограмму становится возможным определять другие объекты этого класса, как в языках Ada и C++. В языке Ada имена также могут быть *перегружены* (overloaded), то есть несколько объектов могут иметь одно и то же имя. До тех пор пока компилятор может определить, на какой именно объект указывает данное имя, не требуется никаких изменений в вызывающей программе. Более подробно перегрузка описана в главе 7.

Тестирование и отладка. Большинство языков имеют различные возможности для облегчения тестирования и отладки. Ниже приведено несколько характерных примеров.

1. *Возможность трассировки.* Prolog, LISP и многие другие интерактивные языки предоставляют программисту возможность отмечать конкретные операторы или переменные для их отслеживания во время выполнения программы. Всегда, когда выполняется помеченный оператор или помеченная переменная принимает новое значение, выполнение программы прерывается и вызывается заданная подпрограмма трассировки (которая, как правило, выводит на печать соответствующую информацию по отладке).
2. *Точки останова.* В интерактивных средах программирования обычно обеспечивается возможность обозначать некоторые места программы как *точки останова*. Когда в процессе выполнения программы достигается точка оста-

нова, выполнение программы прерывается и контроль передается программисту, сидящему за терминалом. Он может проверить и изменить значения переменных и запустить программу заново с момента останова.

3. *Утверждение.* Утверждение — это условное выражение, которое вставляется в программу в виде отдельного оператора, например,

```
assert (X>0 and A=1)or (X=0 and A>B+10).
```

Утверждение определяет, какие соотношения должны выполняться между переменными в данном месте программы. Если включен режим обработки утверждений, компилятор вставляет в откомпилированную программу соответствующий код для проверки указанных условий. Если в процессе выполнения программы переменные не удовлетворяет заданным соотношениям, то выполнение программы прерывается и обработчик исключительных ситуаций выводит на печать некоторое сообщение или предпринимает какое-либо иное действие. После отладки программы утверждения могут быть отключены, так что компилятор не будет генерировать код для их проверки. Впоследствии они могут использоваться как полезные комментарии, облегчающие документирование программы. Эта простая концепция присутствует в нескольких языках, в том числе и в C++.

1.4.2. Среда разработки

Среда разработки (environment framework) состоит из вспомогательных средств и инструментов, отражающих в конечном счете инфраструктуру среды существования программы и используемых для управления разработкой программы. Она обеспечивает программиста необходимыми при разработке программ сервисами, позволяющими организовать хранение данных, быстро разработать графический интерфейс пользователя, обеспечить безопасность и коммуникационную связь с другими программами. Программисты неизбежно используют инфраструктурные сервисы как компоненты при разработке своих программ. Соответственно, иногда языки разрабатываются таким образом, чтобы облегчить доступ к этим сервисам.

Например, программы, написанные в 60-е гг., содержали специальные стандартные процедуры ввода-вывода для обеспечения связи с пользователем. С развитием интерактивных систем стандартным форматом вывода стали окна, выводимые на экран. В настоящее время среды разработки должны включать в себя базовую программу управления окнами (например, Motif, которая использует систему X Windows), а пользовательской программе остается только вызывать специальные функции этой программы для отображения окон, меню, полос прокрутки и выполнения большинства других стандартных действий с окнами. Интерфейсы X Windows являются составной частью среды разработки, которая предоставляет возможность всем использующим ее программам обеспечить для пользователя, работающего с ними за терминалом, некоторый стандартный образец поведения. Такие системы, как Visual Basic и Microsoft Visual Studio предлагают библиотеки сервисов для построения оконных приложений на языках C++, Java и BASIC.

1.4.3. Языки управления заданиями и языки создания процессов

Понятие *управления заданиями* имеет некоторое отношение к среде разработки. В настоящее время, если вы хотите запустить некоторую программу, такую как компилятор C или текстовый процессор, вы наводите указатель мыши на соответствующую картинку или значок на экране и щелкаете на нем мышью. До того как наступила эпоха оконных систем, для запуска программы вам надо было вручную набрать ее имя в командной строке (сегодня это можно сделать в режиме MS-DOS). Еще раньше, в эпоху перфокарт, для запуска программы вначале нужно было вставить перфокарту с именем программы, а затем уже перфокарты, содержащие данные.

Все эти способы позволяют пользователю осуществлять непосредственный контроль над последовательностью действий, необходимых для выполнения работы. Если процесс компиляции программы завершился с ошибками, пользователь может запустить редактор, чтобы исправить текст программы. Если компиляция прошла успешно, пользователь может запустить загрузчик и выполнить программу.

Как было кратко описано в разделе 1.2.3, в 60-е гг., когда функция управления выполнением программ перешла к компьютерам, появилось понятие *процесс*. Оператору не нужно стало ждать успешного выполнения какого-либо этапа программы. Каждая программа стала выдавать *код возврата*, и команда для определения следующего этапа могла проверить код возврата предыдущего этапа и в зависимости от этого кода принять то или иное решение относительно следующего этапа. Таким образом, последовательность этапов — компиляция, загрузка, выполнение первой программы, выполнение второй программы — могла быть загружена в компьютер заранее и операционная система должна была выполнить эту последовательность действий указанным образом. Фирма IBM стала использовать этот подход в своем языке управления заданиями Job Control Language (JCL) для компьютеров серии System 360 в 1963 г.

Операционная система UNIX расширила эту концепцию. Вместо того чтобы просто проверять код возврата предыдущего этапа, язык управления стал выполнять более сложные функции и превратился в более сложную структуру с данными, операциями и операторами. В этом случае объектами данных являются программы и файлы имеющейся компьютерной системы. Таким образом, пользователь мог писать программы, которые связывают между собой различные операции, взятые из других программ. Можно было запрограммировать последовательное выполнение этапов, основываясь на содержании файлов с данными и результатах предыдущих этапов. Это привело к появлению в операционной системе UNIX *командного интерпретатора shell*, имеющего различные разновидности, такие как Bourne shell, C shell, Korn shell и т. д. Файлы с расширением *.bat на персональном компьютере являются простой формой таких командных файлов.

Концепция командного интерпретатора породила много родственных языков. Все они относятся к категории *языков сценариев* или *языков процессов*. В основном это интерпретируемые языки, которые характеризуются тем, что файлы и программы в них рассматриваются как базовые данные. Такие языки, как AWK, Perl и

TCL в течение многих лет использовались для создания сценариев, и последние двадцать лет они являются неотъемлемым атрибутом системного программирования. Однако их популярность резко выросла с появлением Всемирной паутины WWW. Такие языки сценариев имеют большое значение при передаче информации в обоих направлениях между web-браузером пользователя и web-сервером. Позже, в разделе 12.2.2, будет показано, как один из таких языков, а именно Perl, используется для разработки web-программ.

В настоящее время мы находимся в ситуации, когда каждая компания старается создавать продукты «быстрее, лучше, дешевле». Термин *время Интернета* появился для обозначения того, что процесс разработки программного обеспечения должен происходить со скоростью Интернета — мегабиты в секунду. Использование интерпретируемых языков, таких как языки управления заданиями и языки сценариев, позволяет разработчикам *быстро создавать прототипы* приложений. Языки типа Perl позволяют очень быстро строить простые алгоритмы, которые вызывают написанные ранее программные продукты, используя сценарии shell.

1.5. Обзор языка С

Язык С является одним из языков, который тесно связан со своей средой программирования.

История языка. Язык С был разработан в 1972 г. Дэнисом Ритчи (Dennis Ritchie) и Кеном Томпсоном (Ken Thompson) из AT&T Bell Telephone Laboratories. По стилю он похож на ALGOL и Pascal, а также использует свойства языка PL/I. Хотя он и является универсальным языком, компактный синтаксис и эффективность выполнения написанных на нем программ сделали его популярным языком системного программирования.

В конце 60-х AT&T Bell Telephone Laboratories закончила совместный с MIT и GE (General Electric) проект по созданию операционной системы Multics; тем не менее целью Кена Томпсона оставалась разработка удобной операционной системы. Таким образом зародилась идея операционной системы UNIX. Система Multics программировалась на языке PL/I, и хотя он достаточно громоздок, было желание написать новую систему, UNIX, именно на языке высокого уровня. Поскольку Томпсон имел некоторый опыт работы с системным языком BCPL (языком низкого уровня, не имеющим никаких средств динамической поддержки), он разработал новый язык под названием B, который реализовывал минимальное подмножество возможностей языка BCPL для системного программирования («BCPL помещался в 8-килобайтную память компьютера [PDP-7]» [92]). В настоящее время трудно представить, насколько существенным сдерживающим фактором было ограничение объема памяти всего 30 лет назад.

В 1970 г. для проекта UNIX был приобретен компьютер PDP-11 с его огромной памятью в 24 Кбайт. В это время небольшое, но растущее сообщество UNIX почувствовало ограничения языка B. Поэтому в B были добавлены такие понятия, как типы, определения структур и некоторые дополнительные операторы, и новый язык стал известен под именем С.

Хотя С — универсальный язык программирования, он тесно связан с системным программированием. Этот язык впервые был использован для написания ядра операционной системы UNIX и с тех пор был тесно связан с реализациями UNIX. В настоящее время реализации языка С существуют в большинстве компьютерных систем.

В 70-е гг. интерес к языку С проявлялся в основном со стороны университетских кругов, отдававших предпочтение операционной системе UNIX. Когда в 80-х гг. начали появляться коммерческие версии этой системы, их популярность постоянно росла благодаря языку С. В 1982 г. рабочая группа института стандартов ANSI начала работать над стандартом языка С, который увидел свет в 1989 г. [10] и был принят и в качестве международного стандарта (ISO/IES 9899) в 1990 г.

В настоящее время программисты, использующие язык С, — наиболее быстро растущая популяция в программистском мире. Вместе с языками С++ и Java, которые разрабатывались на его основе, он оказывает наибольшее влияние на программирование. Синтаксис и семантика большинства новых языков (таких как Java и Perl) частично основаны на концепциях, характерных для языка С.

Краткий обзор языка. Когда речь идет о языке С, как правило, рассматривается не столько собственно сам язык, определяемый специальной грамматикой, а среда этого языка. Составные части среды программирования языка С таковы.

- *Язык С.* Это относительно небольшой язык с ограниченным набором структур управления и возможностей. (Не забывайте, что он развился из минимального компилятора, который работал на PDP-7, а позже на PDP-11.)
- *Препроцессор С.* Поскольку почти каждый компилятор С включает в себя операторы препроцессора, начинающиеся с символа #, то большинство пользователей даже не подозревают, что они не являются составной частью собственно языка С.
- *Допущения интерфейса С.* В результате использования возможностей языка С возник ряд соглашений. Например, принято, что определения интерфейсов между модулями должны храниться в соответствующем заголовочном файле с расширением .h. В операторе

  ```
  #include "myfcn.h"
  ```

  при определении интерфейса к модулю myfcn используется как С-препроцессор, так и указанное соглашение.
- *Библиотека С.* Многие функции, такие как printf, getchar, malloc, fork и exec, были написаны с помощью интерфейсов С, хотя они не входят в официальное определение языка С. Однако в стандарте ANSI языка С эти функции включены в язык в качестве обязательных библиотечных функций для согласующихся со стандартом компиляторов. Подключение большой библиотеки позволяет расширять с помощью ее функций относительно небольшое ядро языка.

Модуль на языке С состоит из глобальных объявлений и последовательности вызовов функций. Для формирования одной выполняемой программы одновременно загружается несколько модулей. Каждая функция может вызывать другие функции и имеет доступ как к локальным, так и к глобальным данным.

При такой структуре хранение данных осуществляется очень просто. Каждая функция имеет локальную память или локальную активационную запись, которая является динамической и допускает рекурсию; также каждая функция имеет доступ к глобальным переменным. Реально не существует никакой блочной структуры. Каждый отдельный объект данных реализован эффективно. Например, многомерные массивы строятся из одномерных массивов, а индексы одномерных массивов начинаются с 0. Это исключает необходимость применения дескрипторов для вычисления смещения, что, в свою очередь, исключает все сложные вычисления для определения размещения в памяти элементов массива.

В языке С используются указатели, а массивы и указатели эквивалентны, что позволяет программам использовать любой из наиболее подходящих методов для доступа к элементам массивов. Строки реализованы как массивы символов. Эта реализация совершенно прозрачна, поэтому обращаться к строкам можно несколькими способами: как к строкам, как к массивам или (как уже отмечалось) как к указателям на отдельный символ.

В языке С имеется большой набор арифметических операций, которые позволяют писать очень эффективные, а иногда и очень лаконичные программы. Также имеется полный набор структур управления с очень гибкой семантикой, иногда допускающей довольно необычное использование.

Кроме того, в С предусмотрена возможность гибкого определения типов. Тем не менее можно утверждать, что С является одновременно и сильно типизированным, и не сильно типизированным языком. Такая неоднозначность возникает из-за того, что большинство отдельных элементов данных являются подтипами целого типа. Хотя транслятор и обнаружит ошибки, связанные с несоответствием типов, но поскольку большинство элементов данных в конечном счете относятся к целочисленному типу, многие ошибки могут остаться незамеченными.

Язык С изначально был тесно связан с функциональными возможностями операционной системы. В операционной системе UNIX некоторые функции операционной системы (например, функция malloc для динамического выделения области памяти) определены как вызовы соответствующих функций языка С. По соглашению все они определяются в системных заголовочных файлах с расширением .h. Так, например, для вызова malloc из программы на языке С в начале программы должен быть подключен соответствующий файл

```
#include <malloc.h>
```

а при дальнейшем использовании этой функции в программе следует просто писать malloc(StorageSize), указывая в качестве параметра требуемый объем памяти. При компиляции и запуске такой программы определения соответствующих функций (например, malloc) подключаются из библиотеки С.

Компилятор С сначала запускает препроцессор. Такие команды, как #define и #include выполняются в первую очередь, а затем уже транслятором С компилируется вся остальная программа.

При разработке операторов ввода-вывода в качестве образца была взята концепция оператора FORMAT языка FORTRAN, но получившиеся в результате операторы лучше приспособлены для интерактивных программ, чем операторы READ и WRITE языка FORTRAN. Большинство полезных функций определено в системном файле stdio.h, который должен подключаться к любой программе на языке С. Та-

кой подход позволяет легко расширять язык — достаточно написать ряд новых функций для добавления новых функциональных возможностей.

В программе можно использовать комментарии (любой текст, ограниченный символами /*...*/), задаваемые в любом месте, где можно использовать пробел. Поскольку формат текста свободный, то символы продолжения строки не нужны, однако каждый макрооператор препроцессора обязательно должен определяться в одной строке. Тем не менее можно использовать символ (\) для указания, что определение макрооператора препроцессора продолжается в следующей строке.

1.6. Рекомендуемая литература

Во многих книгах конкретные языки программирования рассматриваются на обзорном уровне. В книгах Дершема и Джипинга [35], Лоудена [74], Себесты [99] и Сети [100] предложен альтернативный взгляд на многие аспекты, освещенные в данной книге. Ранний этап истории развития языков программирования хорошо изложен в книгах Саммета [95], [96] и Розена [93]. Под редакцией Вексельблата (Wexelblat) вышел сборник статей, написанных разработчиками многих основных языков, среди которых ALGOL, APL, COBOL, FORTRAN PL/1 [117]. В 1993 г. прошла Вторая конференция по истории языков программирования [4]. В этом издании приведена более полная информация по истории языков Prolog, C, LISP, C++, Ada и Pascal, а также некоторых других, о которых не рассказано в нашей книге. Роль стандартизации языков описана в книге Рада и Берга [90].

Все языки, описанные в этой книге, испытывают влияние как среды программирования, так и операционной среды. Эти факторы особенно сильно повлияли на такие языки, как Ada, C, LISP, ML и Prolog. Описание многих современных вариантов сред программирования приведено в книгах Брауна [23], а также Перри и Кайзера [87]. О способах проверки корректности программ и об ограничениях методов верификации можно прочитать у Абрамса и Зелковица [2]. Статьи на все рассмотренные в этой главе темы можно найти в журналах *IEEE Transactions on Software Engineering*, *ACM Transactions on Software Engineering* и *Software Practice and Experience*.

1.7. Задачи и упражнения

1. Используя критерии, приведенные в разделе 1.3.1, оцените причины популярности того языка, который наиболее активно используется на вашем локальном компьютере. Нужно ли расширить список критериев?
2. Выберите стандартизованный язык, к компилятору которого у вас есть доступ. Напишите программу, которая, будучи не совместимой, все же компилируется и выполняется. Перечислите те нестандартные возможности, которые поддерживает ваш компилятор.
3. Когда выходит стандарт языка (обычно каждые 5–10 лет), сильное влияние на составление нового стандарта оказывает требование совместимости бо-

лее старых программ с реализациями новых версий. Таким образом, при составлении стандарта имеет место некоторое внешнее давление на разработчиков, вынуждающее их включать предыдущую версию в новый стандарт. Для нескольких стандартных определений языков Ada, COBOL и FORTRAN сравните языковые конструкции и перечислите наиболее слабые из них (с вашей точки зрения), наличие которых в данной версии языка объясняется только требованием совместимости с предыдущими версиями.

4. Рассмотрите следующий простой язык. Символы a и b являются именами целых переменных. Каждый оператор может иметь префиксную метку. Определены следующие операции:

a=b	Присваивает переменной a значение переменной b
a=a+1	Прибавляет 1 к значению переменной a
a=a-1	Вычитает 1 из значения переменной a
if a =0 then goto L	Если a = 0, управление передается оператору с меткой L
if a >0 then goto L	Если a > 0, управление передается оператору с меткой L
goto L	Передает управление оператору с меткой L
halt	Завершает выполнение программы

Например, программа, вычисляющая a=a+b, может быть записана следующим образом:

```
L: a=a+1
 b=b-1
 if b>0 then goto L
 halt
```

а) Напишите на этом языке следующие программы:

1) дано a и b, вычислить x = a + b;

2) дано a и b, вычислить x = a × b;

3) дано a, b, c и d, вычислить x = a × b и y = c × d.

б) Обсудите минимальный набор расширений для того, чтобы этот язык было легко использовать. Рассмотрите такие концепции, как подпрограммы, новые операторы, объявления и т. д.

5. Возьмите какую-нибудь программу, написанную вами на императивном языке (например, FORTRAN, C или Pascal), и перепишите ее так, чтобы она стала аппликативной.

6. Возможности языка C позволяют описать некоторое действие несколькими разными способами. Сколько различных операторов вы можете написать на C, чтобы прибавить 1 к переменной x (эквивалентных оператору x=x+1)? Выясните преимущества и недостатки этого аспекта языка C.

7. Описанный в конце раздела 1.4.1 оператор assert может быть реализован или как тест, используемый во время выполнения программы и выполняемый всякий раз, когда в программе доходит очередь до этого оператора, или как некоторое свойство, для которого следует осуществить проверку на истинность в этой точке программы.

а) Обсудите, как можно реализовать каждый из этих подходов. Какие сложности могут возникнуть при реализации каждого из подходов?

б) Когда условное выражение оператора assert будет всегда истинно для каждого метода реализации?

8. Первые компьютеры были сконструированы как электронные калькуляторы для решения численных задач. Подумайте, как бы могли выглядеть языки программирования, если бы первые компьютеры создавались для разных конкретных целей (например, для обработки текстов, управления роботами, игр).

9. Представьте себе, что некоторый новый язык поддерживает три основных типа данных: *целые* (*integer*), *вещественные* (*real*) и *символьные* (*character*). Также он позволяет объявлять массивы (*array*) и записи (*record*) данных. Элементы массивов должны быть одного типа, а записи могут состоять из элементов различных типов. Используйте понятие ортогональности для сравнительного анализа следующих двух новых вариантов этого языка:

   а) Элементы массивов и записей могут быть одного из трех основных типов данных и также могут быть массивами или записями (например, элемент записи может быть массивом).

   б) Элементы массивов и записей могут быть *вещественными* (*real*) или *целыми* (*integer*). Массивы символов называются *строками* (*string*) и обрабатываются специальным образом. Элементы записей могут быть *символьного* типа и массивами, но записи не могут быть элементами массивов. Массивы не могут быть элементами массивов, но можно использовать многомерные массивы для достижения того же результата.

Глава 2. Влияние машинной архитектуры

Как описано в разделе 1.2, ранние языки программирования разрабатывались так, чтобы написанные на них программы могли эффективно работать на дорогостоящих вычислительных машинах. Поэтому программы, написанные на ранних языках (таких как FORTRAN, предназначенный для численных расчетов, или LISP, работающий со списками), транслировались в эффективный машинный код, даже если писать их было достаточно сложно. Теперь ситуация изменилась — машины стали работать гораздо быстрее, их стоимость снизилась, но зато возросла стоимость работы программиста. В настоящее время приоритетное значение имеет создание программ, при написании которых меньше вероятность допустить ошибку, даже если при этом несколько возрастет время их выполнения. Например, определение пользовательских типов данных в ML, объект класса (class) в C++ и спецификация пакета (package) в языке Ada упрощают процесс написания корректно работающих программ, но за это приходится расплачиваться уменьшением скорости выполнения. В этой главе мы постараемся ответить на вопрос, как создаются языки программирования.

При разработке языка программирования учитывается архитектура программного обеспечения, которая, соответственно, оказывает влияние на идеологию создаваемого языка. Это влияние определяется:

1) реальным компьютером, на котором будут выполняться написанные на данном языке программы;
2) моделью выполнения, или виртуальным компьютером, который поддерживает этот язык на реальном компьютере.

2.1. Структура и принципы работы компьютера

Компьютер представляет собой интегрированный набор алгоритмов и структур данных, способный хранить и выполнять программы. Компьютер может быть создан как реальное физическое устройство и состоять из проводов, интегральных схем, монтажных плат и тому подобных элементов — в таком случае мы говорим о *реальном*, или *физическом, компьютере*, называя его просто *компьютер*. Но он также может быть построен с помощью программ, выполняемых на каком-то другом компьютере, — тогда он называется *программно-моделируемым* компьютером. Ре-

ализация языка программирования осуществляется путем создания *транслятора*, который транслирует программы, написанные на данном языке, в программы на машинном языке, которые уже могут непосредственно выполняться на каком-либо компьютере. Этот компьютер может являться как реальным физическим компьютером, так и *виртуальным* компьютером, состоящим частично из аппаратуры и частично из программного обеспечения.

Любой компьютер составляют шесть основных компонентов, функциональность которых тесно взаимосвязана с основными аспектами языка программирования.

1. *Данные*. Компьютер должен обрабатывать различные типы простейших элементов данных и структур данных.
2. *Элементарные операции*. Компьютер должен иметь набор элементарных операций для работы с данными.
3. *Управление последовательностью действий*. Компьютер должен обеспечивать управление последовательностью выполнения элементарных операций.
4. *Доступ к данным*. Компьютер должен предоставлять механизмы управления данными, которые необходимы для выполнения любой операции.
5. *Управление памятью*. Компьютер должен предоставлять механизмы управления распределением памяти для программ и данных.
6. *Операционная среда*. Компьютер должен предоставлять механизмы связи с внешней средой, содержащей программы и данные, подлежащие обработке.

Эти задачи определяют особенности разработки языков программирования. Но прежде, чем рассматривать их в достаточно сложном контексте языков программирования высокого уровня, следует посмотреть на них в более простом контексте реального физического компьютера.

2.1.1. Аппаратные средства компьютера

Архитектура компьютеров весьма разнообразна, но рис. 2.1 иллюстрирует довольно типичную, традиционную организацию большинства компьютеров. *Оперативная память* содержит программы и данные, которые подлежат обработке. Эта обработка выполняется посредством *интерпретатора*, который по очереди декодирует каждую команду на машинном языке и вызывает указанную простейшую операцию вместе с указанными в качестве операндов входными данными. Простейшие операции манипулируют данными, расположенными в оперативной памяти и в быстрых *регистрах*, а также могут осуществлять обмен программами или данными между памятью и внешней операционной средой. Давайте более детально рассмотрим перечисленные выше шесть основных функциональных особенностей компьютера.

Данные. На схеме (рис. 2.1) изображены три основных компонента, предназначенные для хранения данных: оперативная память, быстрые регистры и внешние файлы. *Оперативная память* обычно представляет собой линейную последовательность битов, разделенную на слова фиксированной длины (как правило, слово содержит 32 или 64 бита) или байты, состоящие из 8 битов (4 или 8 байтов на одно слово). *Быстрые регистры* состоят из слов (то есть из последовательно-

стей битов длиной в слово) и могут иметь подполя, к которым возможен прямой доступ. Содержимое регистра может представлять собой данные или адрес в оперативной памяти, по которому находятся эти данные или следующая подлежащая выполнению команда. *Кэш* (быстродействующая буферная память) обычно расположен между быстрыми регистрами и оперативной памятью и является тем механизмом, который ускоряет доступ к данным, хранящимся в оперативной памяти. *Внешние файлы* хранятся на жестком диске, на дискете или компакт-диске. Они обычно состоят из набора записей, каждая из которых является последовательностью битов или байтов.

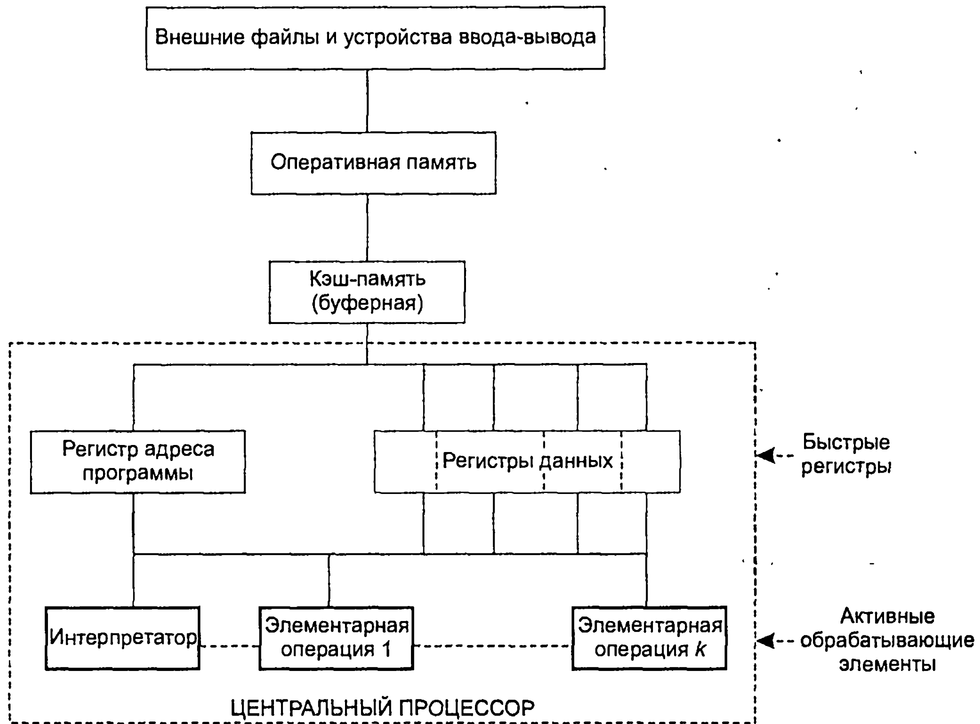


Рис. 2.1. Организация традиционного компьютера

Компьютер располагает некоторым набором встроенных типов данных, которыми можно манипулировать напрямую при помощи простейших операций. Обычный набор таких типов данных включает целые числа, вещественные числа одинарной точности (то есть представленные одним словом), известные как *числа с плавающей точкой*, символьные строки фиксированной длины и строки битов фиксированной длины (эта длина в данном случае равна количеству битов, которые помещаются в одно слово памяти).

Помимо этих очевидных, аппаратно поддерживаемых элементов данных, программы также представляют собой некоторую форму данных. Как и для других встроенных типов данных, в компьютере должно существовать встроенное представление для программ, называемое *представлением на машинном языке* ком-

пьютера. Как правило, представление программы на машинном языке состоит из последовательности ячеек памяти, в каждой из которых содержится одна или более команд. Каждая команда, в свою очередь, состоит из кода операции и набора указателей операндов.

Операции. В компьютере должен присутствовать набор встроенных простейших операций, которым обычно сопоставлены коды операций, используемые в командах машинного языка. В типичный набор входят простейшие операции, производящие арифметические действия (сложение, вычитание, умножение и деление) для каждого из встроенных типов численных данных (целочисленных и вещественных); простейшие операции для проверки различных свойств элементов данных (например, для проверки на равенство нулю, определение знака); простейшие операции для доступа к различным частям элементов данных и их модификации (например, извлечение символа из содержащего его слова или его запись туда, извлечение адреса какого-либо операнда из некоторой команды или его запись туда); простейшие операции для управления устройствами ввода и вывода данных и, наконец, простейшие операции для управления последовательностью выполнения команд (например, безусловный переход к некоторой команде или возврат в некоторую точку программы).

По мере развития традиционных компьютеров (так называемых CISC-компьютеров, или *компьютеров с полным набором команд* — complex instructions set computers) набор доступных команд все более расширялся и становился более мощным механизмом увеличения эффективной скорости работы компьютера. С другой стороны, было обнаружено, что, напротив, увеличить скорость работы компьютера можно и с меньшим числом простейших команд, так как в этом случае упрощается внутренняя логика центрального процессора (ЦП). Такие компьютеры называются RISC-компьютерами (reduced instruction set computers — *компьютеры с сокращенным набором команд*); мы обсудим их подробнее в разделе 11.3.

Управление последовательностью действий. При выполнении программы на машинном языке адрес очередной команды, которая должна выполняться, определяется содержимым специального *регистра программных адресов* (также известного под названием *счетчика команд*), в котором всегда содержится адрес следующей команды. Некоторые простейшие операции могут изменять содержимое этого регистра, чтобы передать управление в другую часть программы, но в действительности этот регистр используется только *интерпретатором*, который и управляет последовательностью выполнения операций.

Интерпретатор играет центральную роль в работе компьютера. Обычно он выполняет простой циклический алгоритм, схематично представленный на рис. 2.2. Во время выполнения каждого цикла интерпретатор получает адрес следующей команды из регистра программных адресов (и увеличивает значение этого регистра, присваивая ему адрес следующей команды), выбирает указанную команду из памяти, декодирует ее, разбивая на код операции и набор указателей операндов, при необходимости выбирает из памяти указанные операнды и вызывает указанную операцию с указанными операндами в качестве аргументов. Простейшие операции могут изменять данные в памяти или в регистрах, обращаться к устройствам ввода-вывода и модифицировать последовательность выполнения операций путем изменения содержимого регистра программных адресов. После выполнения

указанной операции интерпретатор просто возвращается к началу цикла и снова повторяет все описанные действия.

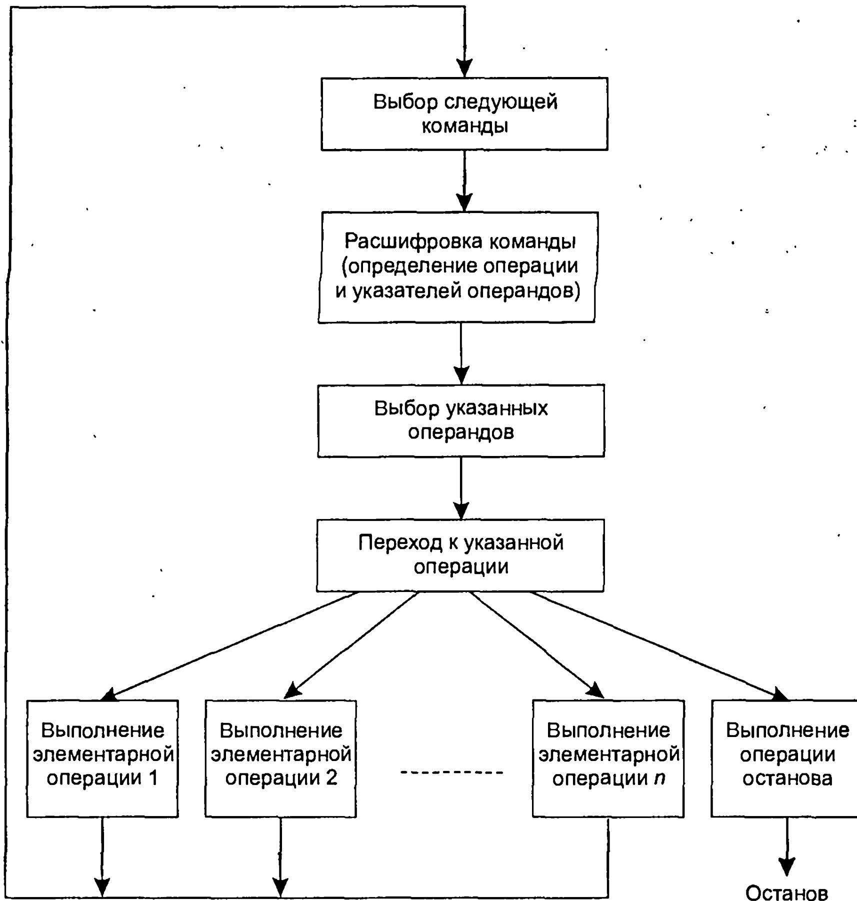


Рис. 2.2. Алгоритм интерпретации и выполнения программы

Доступ к данным. В любой машинной команде помимо кода операции должны быть указаны те операнды, к которым эта операция будет применена. Обычно операнд содержится в оперативной памяти или в рабочем регистре. Компьютер должен обладать средствами для указания операндов и для извлечения значения операнда, находящегося по заданному указателем адресу. Аналогично результат выполнения простейшей операции должен сохраняться в некотором определенном месте памяти. Описанные средства мы называем средствами управления доступом к данным. Традиционная схема заключается в том, что областям памяти ставятся в соответствие некоторые целочисленные значения, которые играют роль адресов этих областей, а специальные операции извлекают по заданному адресу содержимое соответствующих областей памяти (или, наоборот, сохраняют в области памяти с заданным адресом некоторое новое значение). Аналогично регистры часто задаются простым целочисленным адресом.

Управление памятью. Одним из ведущих принципов организации работы компьютера является сведение к минимуму времени бездействия каждой из его составных частей (оперативной памяти, центрального процессора, внешних устройств). Основная трудность, встречающаяся при попытке реализации этого принципа, заключается в том, что в ЦП время на выполнение операций обычно измеряется в наносекундах (например, на выполнение одной типичной операции в современном процессоре требуется 5–10 нс), получение данных из оперативной памяти занимает уже несколько микросекунд (что составляет примерно 50–70 нс), а обращение к внешним источникам требует затрат времени порядка 10–15 мс. В итоге внутренняя скорость работы микропроцессора отличается от скорости доступа к дисковым данным в 1 000 000 раз. Для того чтобы подходящим образом сбалансировать эти скорости, используются различные средства управления памятью.

В простейшем случае (например, в сравнительно дешевом персональном компьютере для домашнего использования) предусмотрены только самые простые средства управления памятью, встроенные в аппаратную часть компьютера. Программы и данные остаются в памяти в течение всего времени выполнения программы, и зачастую в определенный момент времени может выполняться только одна-единственная программа. Хотя процессору и приходится простаивать в ожидании доступности необходимых данных, оказывается более выгодно (в отношении цены) не использовать дополнительные аппаратные средства, ускоряющие выполнение программ.

В операционных системах часто используется *мультипрограммирование* как способ решения проблемы, возникающей из-за различия между скоростью доступа к внешним данным и скоростью работы процессора. Этот способ заключается в том, что пока необходимые для выполнения определенной программы данные считываются с внешних устройств (например, с диска), процессор переходит к выполнению какой-либо другой программы. Для того чтобы в памяти могли сосуществовать несколько различных программ в одно и то же время, сейчас обычной практикой стало включение средств для *страничной организации памяти* (paging) или *динамического перемещения программ* непосредственно в аппаратную часть. Алгоритмы страничной организации памяти позволяют предсказать с определенной вероятностью, какие программы и данные потребуются в ближайшее время, с тем чтобы аппаратные средства компьютера обеспечили процессору возможность максимально быстрого доступа к ним. Программы выполняются до тех пор, пока в оперативной памяти имеется правильная страница с необходимыми командами или данными. Если же в какой-то момент возникает *ошибка из-за отсутствия страницы* (page fault; то есть требуемая команда или данные расположены на странице, отсутствующей в оперативной памяти), тогда операционная система извлечет ее из внешнего устройства хранения данных, в то время как процессор переключится на выполнение другой программы, для которой в оперативной памяти имеются необходимые данные и команды.

Для решения проблемы различия между скоростью доступа к основной памяти и скоростью работы центрального процессора используется *кэш* — быстродействующая буферная память небольшого объема, которая является промежуточным звеном между оперативной памятью и центральным процессором (см. рис. 2.1). Вме-

стимость кэша невелика: от 1 Кбайт (Кбайт — стандартное обозначение для 1024, или 2^{10}, байт) до 512 Кбайт. В нем содержатся данные и инструкции, которые только что были использованы центральным процессором и, следовательно, с большой вероятностью понадобятся в ближайшем будущем. Данные, расположенные в кэше, незамедлительно становятся доступными для процессора, как только в них возникает необходимость; если данные в кэше подвергаются изменениям в результате выполнения программы, то эти измененные данные записываются в оперативную память и скорость доступа к ним уменьшается. Если данные с указанным адресом не содержатся в кэше, то на аппаратном уровне извлекается блок данных, содержащий данные с требуемым адресом вместе с данными, содержащимися в нескольких последующих адресах, которые с наибольшей вероятностью потребуются процессору в ближайшее время.

Кэш позволяет компьютеру работать так, как если бы скорость оперативной памяти была такой же, как скорость центрального процессора. Даже такой небольшой объем кэша, как 32 Кбайт, позволяет увеличить эффективность работы процессора до 95 %, то есть почти до такого уровня, как если бы скорость оперативной памяти совпадала со скоростью центрального процессора.

Операционная среда. Операционная среда компьютера обычно состоит из набора внешних запоминающих устройств и устройств ввода-вывода. Эти устройства являются средством связи компьютера с внешним миром, и любое общение с компьютером возможно лишь через его операционную среду. Между различными типами внешних устройств, составляющих его операционную среду, существуют аппаратные различия, обусловленные различным их назначением или скоростью доступа к ним. Например, существуют быстрые запоминающие устройства (расширенная память), устройства хранения со средней скоростью доступа (гибкие диски и компакт-диски), медленные запоминающие устройства (ленты), а также устройства ввода-вывода (устройства чтения данных, принтеры, мониторы, линии передачи данных).

Альтернативная архитектура компьютера. Аппаратная часть компьютера может быть организована различными способами. Описанную выше архитектуру обычно называют *архитектурой фон Неймана*. Но существуют и другие варианты.

Архитектура фон Неймана. Компьютер фон Неймана называется так в честь математика Джона фон Неймана. Основные принципы устройства этого компьютера были разработаны им в начале 1940-х гг. как часть проекта создания одной из первых электронно-вычислительных машин ENIAC. В варианте фон Неймана компьютер состоял из небольшого *центрального процессора* (ЦП), который выполнял простейшие операции, осуществлял управление последовательностью действий и включал в себя регистры для хранения результатов простейших операций; также имелась оперативная память и была предусмотрена возможность получения, хранения и обмена данными в форме слов между ЦП и оперативной памятью. До сих пор большинство компьютеров создается на основе этой модели, хотя для улучшения качества работы были сделаны такие дополнения, как кэш, виртуальная память и увеличение количества регистров ЦП.

Мультипроцессоры. Как уже было сказано, основная проблема компьютера фон Неймана заключается в большой несбалансированности скоростей доступа к данным, хранящимся на внешних запоминающих устройствах, и в быстрых регистрах

центрального процессора. Альтернативный путь решения этой проблемы заключается в использовании нескольких ЦП в одной системе. Такие *мультипроцессорные* системы применяются уже более 30 лет. Объединение нескольких сравнительно недорогих процессоров с общей оперативной памятью, запоминающими устройствами и аппаратурой ввода-вывода позволяет организовать достаточно эффективную систему. Производительность этой системы возрастает за счет того, что каждый из этих процессоров выполняет какую-то определенную программу, таким образом одновременно выполняется несколько программ.

По большей части эти усовершенствования не оказывают влияния на разработку языков программирования, так как каждый процессор работает только с одной программой, независимой от других. Тем не менее развитие языков программирования и компьютерной архитектуры идет в направлении создания таких систем, в которых программы, выполняемые на нескольких компьютерах, могут взаимодействовать друг с другом. Мы обсудим альтернативные варианты компьютерной архитектуры, которые несколько отличаются от неймановской, в разделе 11.3.2, а также рассмотрим, как эти изменения архитектуры влияют на разработку языков программирования и их трансляцию.

Состояния компьютера. Понимание *статической организации* компьютера в терминах данных, операций, управляющих структур и т. п. представляет только часть общей картины. Для полного понимания необходимо также ясно представлять себе *динамику работы* компьютера во время выполнения программы. То есть нужно знать, каково содержимое различных компонентов памяти в начале выполнения, какова последовательность выполнения различных операций, как различные элементы данных изменяются в процессе выполнения программы и каков конечный результат работы программы.

Для наблюдения за динамикой работы компьютера удобно использовать понятие *состояние компьютера*. Рассмотрим процесс выполнения программы как последовательное прохождение компьютера через ряд состояний. Каждое состояние характеризуется содержимым оперативной памяти, регистров и внешней памяти в определенные моменты времени в процессе выполнения программы. Исходное состояние этих областей памяти определяет соответственно *исходное состояние* компьютера. Каждый шаг при выполнении программы преобразует существующее состояние компьютера в некоторое новое состояние посредством изменения содержимого одной или нескольких указанных областей памяти. Это преобразование называется *сменой состояний*. Когда выполнение программы заканчивается, окончательное содержимое этих областей памяти определяет *конечное состояние* компьютера. Таким образом, выполнение программы можно рассматривать как процесс последовательной смены состояний компьютера.

2.1.2. Программно-аппаратный компьютер

Ранее мы определили компьютер как интегрированный набор алгоритмов и структур данных, способный хранить и выполнять программы. Программы, выполняемые каким-либо компьютером, конечно, написаны на машинном языке этого компьютера. Обычно считается, что машинный язык — это язык низкого уровня с простыми форматами команд и операциями типа «сложить два числа» и «загру-

зить в регистр содержимое области памяти». Но в действительности машинный язык не должен быть непременно языком низкого уровня. Можно выбрать любой язык (например, FORTRAN или ML) и точно определить набор структур данных и алгоритмов, которые задают правила выполнения любой программы, написанной на этом языке. Тогда для организованного таким образом компьютера машинным языком будет выбранный язык программирования. Каждая программа определяет исходное состояние компьютера, а правила выполнения программы определяют последовательность смены состояний, через которые пройдет компьютер в процессе ее выполнения. Результат выполнения программы определяется конечным состоянием компьютера, когда выполнение программы закончится (если это вообще произойдет).

Имея точное определение того, что такое компьютер, всегда можно *сконструировать аппаратное устройство*, машинный язык которого будет в точности тем самым выбранным языком программирования. Это утверждение остается в силе даже для таких языков высокого уровня, как C, Ada и др. (например, компьютер B5500 фирмы Burroughs, упомянутый в разделе 1.2.1). Это утверждение основывается на важном основополагающем принципе компьютерного конструирования: любой точно определенный алгоритм или структура данных могут быть реализованы в аппаратуре. Поскольку компьютер — это просто набор алгоритмов и структур данных, то можно предположить, что его аппаратная реализация возможна независимо от сложности компьютера или используемого машинного языка.

На практике в качестве машинных языков обычно используются языки низкого уровня из тех соображений, что в противном случае (если взять, например, в качестве машинного язык C или Ada) компьютер был бы гораздо сложнее устроен и, следовательно, стоил бы значительно дороже. Кроме того, такой компьютер в большинстве случаев оказался бы гораздо менее гибким средством решения задач программирования, чем компьютер с машинным языком низкого уровня. Аппаратный компьютер с универсальным набором команд низкого уровня, простой, неструктурированной оперативной памятью и набором регистров может быть сравнительно эффективно запрограммирован под любой тип из достаточно широкого класса компьютеров. Об этом пойдет речь в следующих разделах. Иногда создаются компьютеры с машинными языками высокого уровня, но обычно предпочитают другие (не аппаратные) способы реализации этих языков.

Обычной альтернативой строго аппаратной реализации компьютера является *программно-аппаратный компьютер*, который моделируется *микропрограммой*, выполняемой на специальном *микропрограммируемом компьютере*. Машинный язык этого компьютера состоит из набора *микрокоманд* очень низкого уровня, которые обычно реализуют простые передачи данных между оперативной памятью и быстрыми регистрами, непосредственно между самими регистрами и из одних регистров в другие через такие обрабатывающие устройства, как сумматоры и умножители. На основе этого простого набора команд создается специальная микропрограмма, которая определяет цикл интерпретации и простейшие операции желаемого компьютера. Эта микропрограмма моделирует операцию желаемого компьютера на микропрограммируемом *хост-компьютере*. Обычно эта микропрограмма размещается в специально отведенной для нее памяти, доступной только для чтения, на хост-компьютере и выполняется на нем с большой скоростью. Концеп-

ция микропрограммного моделирования, по существу, аналогична технологии моделирования при помощи программного обеспечения, которое мы будем обсуждать в следующем разделе. Различие заключается в том, что хост-компьютер в этом случае специально конструируется для микропрограммирования и способен обеспечить скорость выполнения программ на моделируемом компьютере, сравнимую со скоростью работы соответствующей аппаратной реализации.

Микропрограммируемое моделирование компьютера иногда называется *эмуляцией*. Получившийся в результате компьютер называется *виртуальным компьютером*, поскольку он смоделирован микропрограммой, без которой он бы просто не существовал.

2.1.3. Трансляторы и виртуальная архитектура

Теоретически возможно создать аппаратный или программно-аппаратный компьютер, на котором непосредственно выполнялись бы программы, написанные на любом избранном языке программирования. Таким образом получился бы компьютер с соответствующим машинным языком, например C, Prolog, LISP и т. д. Но создание такого компьютера было бы в большинстве случаев экономически невыгодно. Практические соображения в отношении быстродействия, гибкости и стоимости говорят в пользу создания компьютеров с машинными языками достаточно низкого уровня. Программы же, разумеется, в большинстве случаев пишутся на языках высокого уровня, далеко отстоящих от машинного языка. Поэтому возникает вопрос: как организовать выполнение программ (написанных на языках высокого уровня) на конкретном, имеющемся в распоряжении программиста компьютере, независимо от машинного языка.

Для этой задачи существует два основных решения.

1. *Трансляция* (*компиляция*). Можно сконструировать транслятор, который будет переводить программы, написанные на языках высокого уровня, в эквивалентные программы на машинном языке используемого компьютера. Затем интерпретатор и простейшие операции, встроенные в аппаратную часть компьютера, непосредственно выполняют оттранслированную в машинный код программу. Общий термин *транслятор* используется для обозначения любого языкового процессора, который воспринимает программы на некотором *исходном языке* в качестве входных данных (этот исходный язык может быть как высокого, так и низкого уровня), а на выходе выдает эквивалентные по своей функциональности программы, но уже на другом, так называемом *объектном языке* (который также может быть произвольного уровня). Существуют термины для обозначения некоторых специальных типов трансляторов:

   ✦ *Ассемблер* — это транслятор, у которого объектный язык представляет собой некую разновидность машинного языка какого-либо реального компьютера, а исходный язык, называемый *языком ассемблера*, или просто *ассемблером*, — это, как правило, символическое представление объектного машинного кода. В большинстве случаев каждая инструкция на исходном языке переводится в одну команду на объектном языке.

- *Компилятор* — это транслятор, для которого исходным является язык высокого уровня. Объектный язык близок к машинному языку реального компьютера — это либо язык ассемблера, либо какой-нибудь вариант машинного языка. Например, программы на языке С компилируются, как правило, в программы на языке ассемблера, которые затем транслируются ассемблером в машинный язык.
- *Загрузчик*, или *редактор связей*, — это транслятор, у которого объектный язык состоит из готовых к выполнению машинных команд, а исходный язык почти идентичен объектному. Обычно он состоит из программ на машинном языке в *перемещаемой* форме и таблиц данных, указывающих те точки, в которых перемещаемый код должен быть модифицирован, чтобы стать действительно выполняемым. Например, некоторая подпрограмма P может быть откомпилирована таким образом, что она должна размещаться и использовать область памяти с адресами от 0 до 999, а подпрограмма Q — область памяти с адресами от 0 до 1999. Помимо того, эти подпрограммы могут использовать библиотечные функции, для которых определены адреса памяти от 0 до 4999. Задачей загрузчика является создание единой *выполняемой программы*, в которой используются согласованные адреса, как показано в следующей таблице. Выполняемая программа устроена как единая программа с используемыми адресами от 0 до 7999.

Подпрограмма	Адреса после компиляции	Адреса в выполняемой программе
P	0-999	0-999
Q	0-1999	1000-2999
Библиотека	0-4999	3000-7999

- *Препроцессор*, или *макропроцессор*, — это транслятор, исходный язык которого является расширенной формой какого-либо языка высокого уровня (например, Java или C++), а объектный язык — стандартной версией этого языка. Объектная программа, созданная препроцессором, готова к трансляции и выполнению обычными процессорами исходного стандартного языка. Большинство компиляторов С включает в себя препроцессор, который до начала фазы компиляции сначала преобразует используемые в программе макросы в стандартные операторы языка С.

Трансляция программ с исходного языка высокого уровня на машинный язык часто требует нескольких шагов. Например, нередко программа на C++ сначала транслируется в программу на С, затем компилируется в программу на языке ассемблера, которая далее транслируется в перемещаемый машинный код и, наконец, редактором связей преобразуется в выполняемый машинный код, который уже может быть загружен в память загрузчиком и выполнен. Более того, сама фаза компиляции может включать в себя несколько последовательных этапов, каждый из которых соответствует определенной промежуточной форме программы, прежде чем получится конечная объектная программа.

2. *Программная имитация* (*программная интерпретация*). Вместо того чтобы транслировать программы на языке высокого уровня в эквивалентные программы на машинном языке, можно смоделировать с помощью программ, выполняемых на хост-компьютере, *другой компьютер, для которого машинным языком будет данный язык высокого уровня.* Для этого нужно сконструировать набор программ на машинном языке хост-компьютера, которые моделируют алгоритмы и структуры данных, необходимые для выполнения программ на языке высокого уровня. Другими словами, при помощи программ, выполняемых на хост-компьютере, создается (моделируется) компьютер с машинным языком высокого уровня; этот компьютер мог бы быть создан и аппаратным способом. Подобный метод называется *программной имитацией* (или *программной интерпретацией*) компьютера с машинным языком высокого уровня на хост-компьютере. Для моделируемой машины входными данными являются программы на языке высокого уровня. Основная моделирующая программа, используя алгоритм интерпретации, аналогичный представленному на рис. 2.2, декодирует и выполняет каждый оператор заданной программы в соответствующей последовательности и производит определяемый интерпретируемой программой вывод результирующих данных.

   В таком случае мы говорим, что хост-компьютер создает *виртуальный компьютер*, моделирующий язык высокого уровня. Когда хост-компьютер выполняет программу на языке высокого уровня, сложно сказать, выполняется ли программа непосредственно аппаратной частью компьютера или сначала она преобразуется в программу на машинном языке низкого уровня, а затем уже выполняется.

Отметим разницу между трансляцией и программной интерпретацией. И для транслятора, и для программного интерпретатора входными данными являются программы на языке высокого уровня. Но в результате трансляции получается просто программа на объектном языке, которая затем должна выполняться аппаратным интерпретатором для этого объектного языка, а программный интерпретатор непосредственно выполняет введенную программу. Если проследить за обработкой введенной программы транслятором и программным интерпретатором, то окажется, что транслятор обрабатывает операторы, входящие в программу, в порядке их фактического ввода, а интерпретатор следует логике управления выполняемой программы. Обычно транслятор обрабатывает каждый оператор только один раз, в то время как интерпретатор может обрабатывать некоторые операторы многократно (если, например, они являются частью цикла) или полностью игнорировать другие (если на них не будет передано управление).

Но чистая трансляция и чистая интерпретация — это две крайности; на практике обычно эти технологии используются вместе, дополняя друг друга. Чистая трансляция используется достаточно редко. Это происходит, как правило, если исходный язык достаточно близок к машинному языку, как в случае с ассемблером. Так же редки случаи использования чистой интерпретации — она применяется для интерактивных языков и для языков управления операционными системами. Чаще всего для реализации языка на компьютере используется комбинированный подход, как показано на рис. 2.3. Сначала программа транслируется из своей исход-

ной формы в форму, более удобную для выполнения. Обычно это делается с помощью нескольких независимых частей программы, называемых *процедурами* или *подпрограммами*. На этапе загрузки эти независимые части объединяются с набором программ поддержки выполнения (run-time support routines), реализующих программно-моделируемые операции, что приводит к созданию выполняемой формы программы, операторы которой декодируются и выполняются посредством их интерпретации.

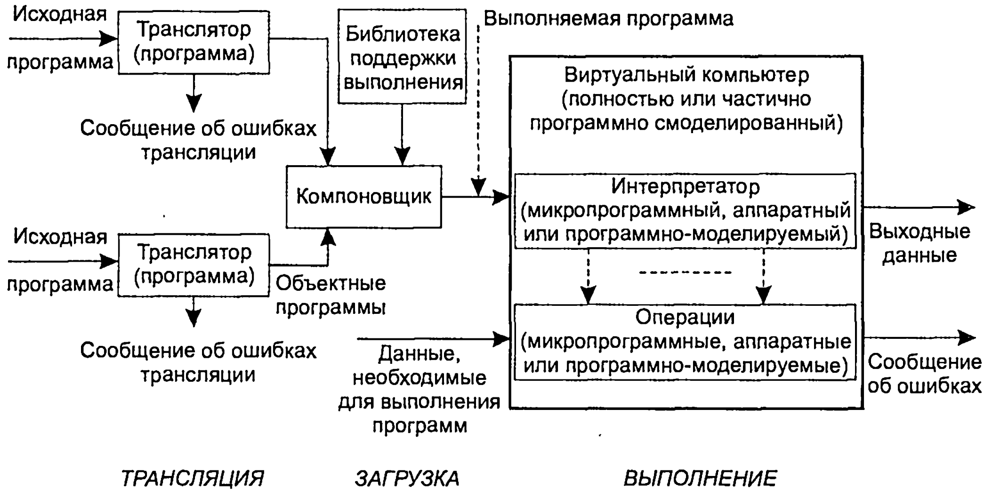


Рис. 2.3. Структура типичной реализации языка программирования

При реализации языка программирования и трансляция, и интерпретация имеют свои преимущества. Некоторые структуры в программах лучше транслировать в более простые формы еще до выполнения, другие лучше оставить в первоначальном виде и обрабатывать по мере надобности в процессе выполнения. Те операторы исходной программы, которые выполняются неоднократно (например, операторы в теле цикла или внутри подпрограммы, которая вызывается более одного раза), эффективнее транслировать. Для выполнения таких операторов часто требуется сложный процесс декодирования, с помощью которого определяются требуемые операции и их операнды. Каждый раз при выполнении операторов этот процесс или большая его часть выполняется заново. Так, если какой-то оператор должен быть выполнен 1000 раз, то и один и тот же процесс его декодирования должен повториться 1000 раз. Если же этот оператор оттранслирован в более простой для декодирования код (например, в последовательность машинных команд), то сложный процесс декодирования происходит только один раз при транслировании, а потом в каждом из 1000 случаев повторного выполнения этого оператора требуется декодировать уже более простой код.

Главным недостатком трансляции является потеря информации о программе. Если данные исходной программы транслируются в последовательность адресов на машинном языке и в программе имеется ошибка (например, один из этих объектов данных делится на 0), часто бывает трудно определить, какой из операторов программы на исходном языке выполнялся и какие объекты данных использова-

лись в нем. При интерпретации вся эта информация остается доступной для пользователя. Кроме того, поскольку оператор на языке высокого уровня содержит гораздо больше информации, чем команда машинного языка, при трансляции размер программы значительно возрастает, то есть программа на машинном языке занимает в памяти гораздо больше места, чем исходная программа.

Интерпретация имеет свои достоинства, почти прямо противоположные перечисленным для трансляции. Операторы остаются в своей исходной форме до тех пор, пока они не понадобятся при выполнении, тем самым не расходуется память для хранения нескольких копий длинных последовательностей машинных команд; все необходимые команды достаточно сохранить один раз в моделирующей программе (интерпретаторе). Однако за это приходится расплачиваться необходимостью многократно декодировать один и тот же оператор, если он, например, встречается в цикле или часто вызываемой подпрограмме.

Ключевой вопрос реализации языка программирования заключается в том, какое представление имеет программа во время ее выполнения на реальном компьютере. Иначе говоря, является этот язык машинным языком данного компьютера или нет? В зависимости от ответа на этот вопрос языки (вернее, их реализации) делятся на *компилируемые* и *интерпретируемые*.

1. *Компилируемые языки.* Языки C, C++, FORTRAN, Pascal и Ada принято считать компилируемыми. Это означает, что программы, написанные на этих языках, транслируются в машинный код данного компьютера перед началом выполнения. Программная интерпретация при этом ограничивается только интерпретацией набора *программ поддержки выполнения*, которые моделируют элементарные операции исходного языка, не имеющие близкого аналога в машинном языке. Транслятор компилируемого языка является обычно довольно большой и сложной программой, и при трансляции основное значение имеет создание максимально эффективных с точки зрения их выполнения программ.
2. *Интерпретируемые языки.* Языки LISP, ML, Perl, Postscript, Prolog и Smalltalk часто реализуются с использованием программного интерпретатора. При такой реализации транслятор выдает не машинный код используемого компьютера, а некую промежуточную форму программы. Эта форма легче для выполнения, чем исходная программа, но все же она отличается от машинного кода. Процесс интерпретации для выполнения полученной программы должен быть реализован программным образом, поскольку аппаратный интерпретатор в данном случае непосредственно применить невозможно. Использование программного интерпретатора обычно приводит к относительно медленному выполнению программы. Трансляторы интерпретируемых языков обычно представляют собой довольно простые программы, основная сложность реализуется в программном обеспечении процесса интерпретации.

Развитие Всемирной паутины WWW и появление языка Java внесли некоторые изменения в описанную схему. Язык Java похож скорее на Pascal и C++, чем на LISP, но в большинстве случаев он реализуется как интерпретируемый язык. Компилятор Java вырабатывает промежуточный набор *байт-кодов* (bytecodes) для

виртуальной машины Java. Основной причиной неэффективности web-приложений является потеря времени при передаче по сети затребованных пользователем страниц, а не выполнение программ на хост-компьютере. Передача байт-кодов на локальный компьютер (даже если он медленнее, чем web-сервер) выгоднее в отношении временных затрат, чем передача результатов выполнения программы на web-сервере. Однако web-сервер не в состоянии предугадать машинную архитектуру хост-компьютера. Поэтому браузер создает виртуальную машину Java, которая и выполняет стандартный набор байт-кодов Java.

2.2. Виртуальные компьютеры и время связывания

В предыдущем разделе компьютер был определен как интегрированный набор алгоритмов и структур данных, способный хранить и выполнять программы. Мы рассмотрели, каким образом можно сконструировать требуемый компьютер.

1. Путем *аппаратной реализации*, когда структуры данных и алгоритмы непосредственно представлены физическими устройствами.
2. Путем *программно-аппаратной реализации*, когда структуры данных и алгоритмы представлены микропрограммами, которые выполняются на соответствующем микропрограммируемом аппаратном компьютере.
3. Как *виртуальный компьютер*, то есть путем моделирования структур данных и алгоритмов программами, написанными на каком-либо другом языке программирования.
4. Посредством *комбинации перечисленных методов*, когда некоторые части требуемого компьютера реализованы аппаратным способом, другие — программно-аппаратным, третьи — при помощи программного моделирования, то есть виртуальным способом.

При реализации языка программирования структуры данных и алгоритмы, задействованные при выполнении программы, определяют некоторую вычислительную машину (компьютер). Поскольку этот компьютер всегда (хотя бы частично) моделируется программным образом (как в случае с программно-аппаратной реализацией компьютера), он называется *виртуальным компьютером, определяемым реализацией языка*. Машинным языком этого виртуального компьютера будет являться выполняемая программа, выдаваемая транслятором этого языка. Если язык компилируемый, эта программа может состоять из машинных команд реального компьютера; она может быть представлена произвольной структурой данных, если язык интерпретируемый. Структурами данных этого виртуального компьютера являются структуры данных, которые используются во время выполнения программы. Примитивные операции — это те операции, которые действительно выполняются аппаратным компьютером при выполнении программы. Аналогично структуры управления последовательностью действий, данными и памятью — это те же структуры, которые используются при выполнении программ, независимо от того, поддерживаются ли они аппаратной частью, микропрограммами или смоделированы программным образом.

2.2.1. Виртуальные компьютеры и реализация языка

Если бы языки программирования определялись в терминах своих виртуальных компьютеров, так что каждый язык ассоциировался бы с некоторым однозначно определенным и понимаемым виртуальным компьютером, то описание семантики языка в терминах виртуального компьютера было бы очевидным. Но в действительности языки обычно определяются путем индивидуального задания семантики для каждой своей синтаксической конструкции, поэтому определение виртуального базового компьютера осуществляется только неявным образом. Каждый раз, когда язык программирования реализуется на новом компьютере, разработчик видит несколько (или совершенно) иной виртуальный компьютер в определении того же самого языка. Таким образом, две различные реализации одного и того же языка программирования могут использовать различные структуры данных и различные наборы операций, особенно это относится к структурам данных и операциям, которые скрыты в программном синтаксисе. Каждый разработчик имеет достаточную широту в определении структур виртуального компьютера, которые составляют основу для конкретной реализации языка.

Когда язык программирования реализуется на каком-либо конкретном компьютере, в первую очередь требуется определить виртуальный компьютер, который представляет собой интерпретацию семантики языка. Затем этот виртуальный компьютер конструируется на основе возможностей аппаратной части и программных элементов, предоставляемых конкретным базовым компьютером. Например, если в виртуальном компьютере присутствуют операция целочисленного сложения и операция извлечения квадратного корня, то разработчик может выбрать следующий вариант. Для реализации целочисленного сложения можно использовать аналогичную операцию, непосредственно встроенную в аппаратную часть базового компьютера, а извлечение квадратного корня можно смоделировать программным образом в виде подпрограммы, вычисляющей квадратные корни. Можно привести пример, иллюстрирующий различные возможности реализации структур данных. Если в виртуальном компьютере имеется простая целочисленная переменная X, можно реализовать ее, непосредственно выделив область памяти, в которой хранится значение этой переменной. Но можно сделать иначе — выделить область памяти, в которой будет содержаться информация о типе данных, хранящихся в переменной (в нашем случае целый тип), и указатель на другую область памяти, в которой содержится фактическое значение переменной X. Организация и структура реализации языка складываются из множества подобных решений, которые принимает разработчик с учетом аппаратных и программных возможностей базового компьютера и стоимости их использования.

Разработчик также должен точно решить, что именно должно быть сделано в процессе трансляции программы, а что — во время выполнения. Часто бывает так, что использование какого-то конкретного способа представления структуры данных или операции виртуального компьютера во время выполнения программы возможно, только если во время трансляции были проделаны определенные действия, необходимые для создания структуры во время выполнения программы. Если разработчик решит сделать транслятор более простым, опустив эти действия,

тогда во время выполнения могут потребоваться различные представления структуры.

Таким образом, различия в реализациях одного и того же языка объясняются следующими тремя факторами.

1. Разные разработчики могут иметь различные мнения о том, как должен быть устроен виртуальный компьютер, неявно определяемый данным языком программирования.
2. Различные базовые компьютеры, на которых реализуется язык, предоставляют различные возможности.
3. Каждый разработчик выбирает собственные решения для реализации элементов виртуального компьютера, используя возможности, предоставляемые базовым компьютером, и для конструирования транслятора таким образом, чтобы он поддерживал выбранные решения для представления виртуального компьютера.

2.2.2. Иерархия виртуальных компьютеров

Виртуальная машина, используемая программистом для создания приложений, состоит фактически из *иерархии виртуальных компьютеров*. В основании этой иерархии, разумеется, должен находиться реальный компьютер. Но обычно программисты крайне редко имеют дело непосредственно с ним. Этот реальный компьютер последовательно преобразуется слоями программного обеспечения (или микропрограммами) в виртуальную машину, которая может значительно отличаться от реального компьютера. Вторым слоем (или третьим, если микропрограммы формируют второй слой) является *операционная система*, представляющая собой сложный набор программ.

Обычно операционная система моделирует некоторое количество новых элементарных операций и структур данных, которые не обеспечены непосредственно аппаратной частью компьютера (например, структуры внешних файлов или функции определения времени). Помимо того, из виртуального компьютера, определяемого операционной системой, исключаются и становятся недоступными для пользователя некоторые элементарные аппаратные операции (например, простейшие операции ввода-вывода, выключения компьютера, мультипрограммирования и организации мультипроцессорной работы). Обычно разработчик, который занимается реализацией языка высокого уровня, имеет дело именно с таким, определяемым операционной системой виртуальным компьютером. В процессе этой реализации разработчик создает новый слой программного обеспечения, которое выполняется на этом виртуальном компьютере и моделирует операции виртуального компьютера для языка высокого уровня. Кроме того, он создает транслятор для языка высокого уровня, переводящий программы пользователя на машинный язык виртуального компьютера.

Фактически описанная иерархическая система не заканчивается на реализации языка высокого уровня. Программы, создаваемые программистом, добавляют следующие уровни к этой иерархии. На рис. 2.4 представлена иерархия виртуальных компьютеров, как она выглядит с точки зрения пользователя WWW.

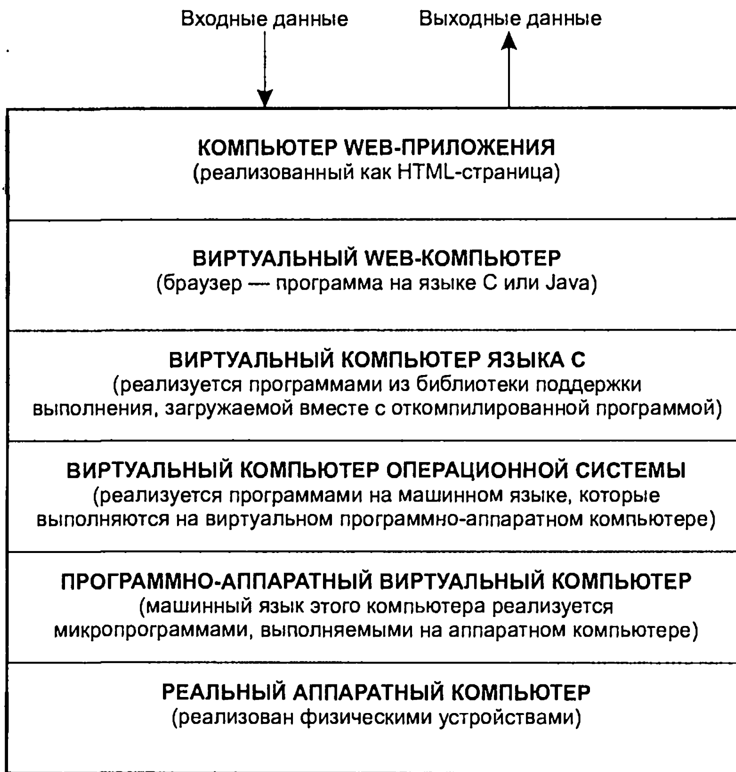


Рис. 2.4. Иерархия виртуальных компьютеров для web-приложения

Над виртуальным компьютером языка С, создаваемым компилятором этого языка, программист с использованием языка С (или другого, ему эквивалентного) строит программу, называемую web-браузером. Этот браузер создает виртуальный web-компьютер, который способен обрабатывать основные структуры данных WWW, гиперссылки (URL) для перехода к другим web-сайтам, основные элементы языка HTML для отображения web-страниц, а также выполнять интерактивные программы (апплеты) для пользователей браузера. Самый верхний уровень в этой конкретной иерархии занимает компьютер web-приложения, реализованный как web-страница. Web-разработчик использует свойства виртуального web-компьютера (HTML, апплеты и т. п.) для предоставления пользователям возможности работать с этой web-страницей.

Центральная концепция, неявно присутствующая в приведенных рассуждениях, заключается в *эквивалентности программы и данных*. Мы привыкли воспринимать одни типы объектов в программировании как программы, а другие — как данные. Часто это интуитивное разделение оказывается полезным, но, как следует из вышеприведенных рассуждений, оно скорее иллюзорное, а не объективно существующее. Например, тот объект, который в каком-то контексте является программой, в другом может представлять собой некую структуру данных. Например, задающий некоторую web-страницу код HTML является структурой данных для виртуального web-компьютера, который обрабатывает эти данные и определяет,

как следует располагать на странице информацию. С другой стороны, для web-разработчика, использующего HTML для обработки той информации, которую он хочет поместить на страницу, этот код является программой.

Область применения принципа эквивалентности данных и программ можно расширить. В языках типа C и FORTRAN выполняемые программы обычно хранятся отдельно от тех данных, которые используются в этих программах. В других языках, таких как Prolog или LISP, такого разделения не существует. Программы и данные хранятся вместе и различие между ними проявляется только в процессе выполнения.

2.2.3. Связывание и время связывания

Если не ставить задачу быть предельно точными, можно сказать, что *связывание* элемента программы с конкретной характеристикой или свойством — это просто выбор определенного свойства из некоторого ряда допустимых свойств. Момент времени, когда при составлении или выполнении программы происходит это связывание, называется *временем связывания* данного свойства с данным элементом. В языках программирования существует множество различных типов связывания, так же как и множество вариантов для времени связывания. Кроме того, в понятия связывания и времени связывания мы включаем свойства элементов программ, которые устанавливаются либо определением языка, либо его реализацией.

Классификация времени связывания

Хотя не существует простого способа классифицировать различные типы связывания, все же можно выделить несколько классов времени связывания. В основе этой классификации лежит наше предположение о том, что обработка программы независимо от языка всегда включает в себя фазу трансляции, за которой следует выполнение оттранслированной программы.

1. *Во время выполнения программы.* Связывание часто происходит во время выполнения программы. Сюда входят связывание переменных с их значениями, а также (во многих языках) связывание переменных с конкретными областями памяти. Здесь можно выделить две важные подкатегории:
   - *При входе в подпрограмму или блок.* В большинстве языков связывание допустимо только при входе в подпрограмму или блок при выполнении программы. Например, в языках C и C++ связывание формальных параметров с фактическими и связывание формальных параметров с определенными областями памяти происходит только во время входа в подпрограмму.
   - *В произвольных точках программы во время ее выполнения.* Некоторые типы связывания могут происходить в произвольных точках программы во время ее выполнения. Наиболее важным примером этого является связывание переменных с их значениями путем присваивания. В таких языках, как LISP, ML и Smalltalk допустимо также связывание имен с областями памяти в произвольный момент при выполнении программы.

2. *Во время трансляции (компиляции).* Можно выделить три класса связываний, происходящих во время трансляции.
   - *Связывание по выбору программиста.* При написании программы программист принимает множество решений по вопросам выбора типов и имен переменных, структуры элементов программы и т. п. Эти решения определяют связывания, происходящие при трансляции. Транслятор данного языка использует эти связывания для определения окончательной формы объектной программы.
   - *Связывание по выбору транслятора.* В некоторых случаях связывание определяется транслятором языка без непосредственного участия программиста. Например, относительное расположение объекта данных в области памяти, отведенной под некоторую процедуру, обычно задается транслятором без какого-либо вмешательства со стороны программиста. Хранение массивов данных и создание (при необходимости) дескрипторов массивов — это также прерогатива транслятора. Для различных реализаций одного и того же языка это связывание может происходить по-разному.
   - *Связывание по выбору загрузчика.* Обычно программа состоит из нескольких подпрограмм, которые должны быть объединены в одну выполняемую программу. Как правило, транслятор связывает переменные с адресами относительно областей памяти, определенных для каждой подпрограммы. Но эти области должны быть размещены в физической памяти с действительными адресами компьютера, на котором программа будет выполняться. Это происходит во *время загрузки*, также называемое *временем редактирования связей.*
3. *Время реализации языка.* Некоторые аспекты определений языка в рамках некоторой его реализации могут быть одними и теми же для всех выполняемых программ, однако они могут различаться в других его реализациях. Например, часто представление чисел и арифметических операций определяется тем, как эти операции реализованы с помощью аппаратной части базового компьютера. Если программа написана на языке, использующем возможности, определения которых были заданы при его реализации, то эта программа не обязательно будет работать в другой реализации этого же языка; хуже того, программа может работать, но выдавать неверный результат.
4. *Время определения языка.* В основном структура языка формируется во время определения языка на основе спецификации альтернатив, доступных программисту при написании программы. Например, возможные альтернативные формы операторов, типы структур данных, программные структуры и т. п. определяются, как правило, во время определения языка.

Чтобы проиллюстрировать многообразие связывания и времен связывания, рассмотрим следующий простой оператор присваивания:

$$X = X + 10.$$

Предположим, что этот оператор появился в некоторой программе, написанной на гипотетическом языке L. Можно исследовать типы связывания и времена связывания следующих элементов этого оператора.

1. *Набор возможных типов для переменной X.* Обычно для переменной в программе определен какой-либо тип данных — это может быть integer (целочисленный), real (вещественный), Boolean (булев) или какой-то иной тип. Множество возможных типов для переменной X обычно устанавливается при определении языка (например, могут допускаться только типы integer, real, Boolean, set (множество) и character (символ)). Может быть и так, что язык допускает определение новых типов переменных в каждой программе, как, например, в C, Java и Ada. В таком случае множество возможных типов для переменной X устанавливается во время трансляции программы.
2. *Тип переменной X.* Конкретный тип данных, ассоциированный с переменной X, часто устанавливается во время трансляции при помощи явного объявления в программе: например, в языке C для того, чтобы задать X как переменную вещественного типа, используется объявление float X. В таких языках, как, например, Perl или Smalltalk, тип данных переменной X определяется во время выполнения программы, в тот момент, когда переменной присваивается значение определенного типа. В этих языках переменная X в одном месте может содержать целое значение, а в другом месте той же программы — строку символов.
3. *Множество возможных значений для переменной X.* Если тип данных для переменной X определен как вещественный (real), то при выполнении программы X может принимать некоторые значения из определенного множества — множества последовательностей битов, представляющих в данном аппаратном компьютере вещественные числа. Множество возможных значений для переменной X зависит от того, какие вещественные числа допустимы в данном виртуальном компьютере, определяемом языком программирования. Обычно это множество совпадает с множеством вещественных чисел, представление для которых предусмотрено в базовом аппаратном компьютере.
4. *Значение переменной X.* В любой момент выполнения программы переменная X связана с каким-либо значением. Это то значение, которое присваивается данной переменной при выполнении программы. Операция присваивания $X = X + 10$ меняет значение переменной X, заменяя старое значение новым, которое больше исходного на 10.
5. *Представление константы 10.* Целое число 10 представлено одновременно как константа в тексте программы (с использованием строки 10) и как некая последовательность битов во время выполнения программы. Выбор десятичного представления этого числа (то есть символа 10 для десяти) обычно происходит во время *определения языка программирования*, а выбор конкретной последовательности битов для представления этого числа при выполнении программы происходит во время реализации языка.
6. *Свойства операции «+».* Выбор символа «+» для обозначения операции сложения происходит во время *определения языка.* Однако в зависимости от контекста этот символ может обозначать *вещественное сложение*, *целочисленное сложение*, *комплексное сложение* и т. п. В компилируемом языке принято определять операцию, представляемую символом «+», во время

компиляции. Роль механизма, выполняющего это связывание, обычно играет механизм определения типов переменных: если X — целое число, то символ «+» обозначает в данном контексте целочисленное сложение; если X — вещественное число, то символ «+» обозначает вещественное сложение и т. д.

Подробное определение операции, обозначенной символом «+», может зависеть также от аппаратной части базового компьютера. Если в нашем примере X будет иметь значение 2^{49}, то на некоторых компьютерах для $X + 10$ не будет возможности представить результат этого сложения.

Подводя итог, можно сказать, что для языка, подобного C, символ «+» связывается с набором операций сложения *во время определения языка*. Затем во время реализации языка определяется каждый элемент из этого набора, во время трансляции программы каждое конкретное употребление символа «+» в программе связывается с конкретной операцией сложения, а конкретное значение результата операции сложения определяется уже во время выполнения программы.

Важность времени связывания

В следующих главах мы займемся сравнительным анализом языков программирования. Во многих случаях в основе различия между языками лежат различия во временах связывания. Мы постоянно будем задавать вопрос: когда именно происходит связывание — во время трансляции или во время выполнения? Многие важные и тонкие отличия между языками заключаются именно в различиях во временах связывания. Например, почти для всех языков допустимы числа в качестве данных и предусмотрены арифметические операции с числами. Но не все языки одинаково хороши для создания программ, в которых требуется выполнять большое количество арифметических операций. Например, и ML, и FORTRAN позволяют определять массивы чисел и манипулировать этими массивами. Но для решения задач, связанных с большими числовыми массивами и большим количеством вычислений, было бы весьма неблагоразумно использовать ML при наличии языка FORTRAN. Если мы попытаемся отыскать причину превосходства FORTRAN, сравнивая характеристики FORTRAN и ML, то в конце концов придем к выводу, что в данном случае лучше использовать FORTRAN, так как в ML большинство связываний происходит во время выполнения программы, а в FORTRAN — во время трансляции. Таким образом, при выполнении программы на ML большая часть времени будет уходить на создание и уничтожение связей. В такой же программе, написанной на FORTRAN, основная часть связей устанавливается единожды во время трансляции, а при выполнении программы потребуется лишь незначительное количество связываний. В итоге программа на FORTRAN будет работать значительно эффективнее.

С другой стороны, можно задать такой вопрос: почему FORTRAN не столь гибок в обработке строковых переменных и массивов, как ML? Ответ опять-таки сводится к различию во времени связывания. Поскольку связывание в FORTRAN происходит по большей части во время трансляции, до того как становятся известны входные данные, на FORTRAN трудно писать программы, которые могли бы приспосабливаться во время выполнения к большому количеству различных ситуаций, зависящих от входных данных. Например, размер строк и тип переменных

в FORTRAN должны быть определены во время трансляции. В ML подобное связывание может быть отложено до момента выполнения программы, когда будут известны входные данные и определятся наиболее подходящие связывания для конкретных входных данных.

В языках, подобных FORTRAN, большая часть связываний происходит на этапе трансляции. Эта ситуация носит название *раннего связывания.* Если же связывания происходят в основном при выполнении программы, как в ML или в HTML, то говорят о *позднем связывании.*

Сравнительные преимущества и недостатки раннего и позднего связывания связаны с противоречием между гибкостью и эффективностью. Языки, для которых эффективность выполнения программ играет решающую роль (например, FORTRAN, Pascal и C), обычно конструируются таким образом, чтобы максимально возможное количество связываний выполнялось во время трансляции. Если же определяющее значение имеет гибкость (как в LISP и ML), большая часть связываний откладывается до момента выполнения программы, чтобы обеспечить возможность учета входных данных, от которых зависят связывания. Если же требуется создать язык, который был бы одновременно и эффективным, и гибким (примером чего служит Ada), следует обеспечить возможность управления временем связывания.

Время связывания и реализация языка

В определении языка обычно не накладывается строгих определений на время связывания. При определении языка предполагается, что какое-то конкретное связывание будет сделано, например, во время трансляции, но окончательно это определяется только при реализации языка. В частности, Pascal устроен так, что тип переменных определяется во время компиляции; но в какой-либо конкретной реализации языка Pascal тип переменных может проверяться только при выполнении программы. То есть хотя в определении Pascal предполагается, что проверка типов переменных происходит во время компиляции, это не является строгим требованием. Вообще говоря, в определении языка указывается самое раннее время, когда в процессе обработки программы может произойти связывание, а при реализации языка связывание фактически может быть отложено до более позднего времени. Тем не менее обычно в различных реализациях данного языка конкретное связывание происходит в одно и то же время. Если язык был определен таким образом, чтобы большинство связываний происходило во время компиляции, то откладывание этих связываний до момента выполнения программы, вероятнее всего, просто уменьшит эффективность работы и не принесет выигрыша в гибкости, как показано в приведенном выше примере с Pascal.

Следует, однако, обратить внимание на то, что часто кажущиеся незначительными изменения в языке могут привести к серьезному изменению времени связывания. Например, введение в FORTRAN 90 рекурсии изменяет время связывания многих важнейших свойств этого языка. Поскольку время связывания зависит от реализации языка, то важно знать, какая конкретная реализация используется. При использовании локальной реализации языка программист должен принимать во внимание времена связывания в этой конкретной версии. Являются ли они обычными или локальные изменения языка привели к их модификации?

2.2.4. Обзор языка Java

История. Разработка языка Java началась в 1991 г. Группа программистов (Green Team) из компании Sun Microsystems под руководством Джеймса Гослинга (James Gosling) занималась разработкой языка для использования в цифровых бытовых устройствах[1]. Летом 1992 г. была создана рабочая версия, но группа опередила свое время, так как в тот момент промышленность еще не была готова к использованию этого языка.

В 1993 г. появился web-браузер Mosaic, что привело к распространению Интернета, вышедшего из стен академических лабораторий, по всему миру. Программисты из группы Green Team сразу же осознали, какую роль может сыграть созданный ими язык для расширения возможностей web-браузеров.

При помощи браузера Mosaic, в котором использовались адреса URL для навигации в сети и код HTML для отображения web-страниц, пользователь мог отыскивать нужные ему страницы и загружать информацию с удаленных сайтов. (Эта тема рассматривается в главе 12.) Тем не менее оставались нерешенными три проблемы, ограничивающие возможность доступа пользователей к WWW.

1. Скорость передачи на компьютер пользователя в 1993 г. ограничивалась примерно 33 000 бит/с (сейчас она составляет примерно 50 000 бит/с).
2. Если какой-либо сайт был достаточно популярен, то при одновременном обращении к нему большого количества пользователей скорость его работы значительно уменьшалась.
3. Для передачи по сети разных типов web-объектов (текстового документа, графического, аудио- или видеообъекта) требовался отдельный протокол, поддерживаемый каждым web-браузером. Новый формат объектов мог быть использован только после того, как соответствующий протокол был включен во все браузеры.

Пока пользователь ждал появления очередной порции информации с сервера, его компьютер фактически простаивал без дела. Чтобы обойти первые два из перечисленных ограничений, было предложено некоторую часть обработки информации перенести на компьютер пользователя, тем самым разгрузив web-сервер. Этого можно было достичь посредством небольшой программы-приложения, которая загружается с web-сервера и выполняется на локальном компьютере, за счет чего сервер может обслуживать большее количество пользователей одновременно. Такая программа называется апплет (applet). Аналогично можно загрузить с web-сервера на локальный компьютер апплет, позволяющий обрабатывать новый тип протокола, необходимый для передачи определенного web-объекта.

По мнению программистов компании Sun, разработанный ими язык мог бы стать ценным дополнением к web-технологиям. Но для эффективности его использования требовалось выполнение нескольких условий.

1. *Независимость от машинной архитектуры.* Web-сервер не имеет информации о том, на каком компьютере установлен браузер пользователя. Для ре-

[1] Этот язык известен под названием Oak. — *Примеч. науч. ред.*

шения этой задачи был сконструирован виртуальный компьютер Java, и апплет мог быть скомпилирован в виде последовательности байт-кодов для данного виртуального компьютера. Таким образом, апплеты Java могли выполняться на любом браузере, в который встроена виртуальная машина Java.

2. *Безопасность.* Для того чтобы язык Java стал общепринятым языком web-программирования, необходимо было обеспечить выполнение с его помощью требований безопасности, имеющих решающее значение для пользователей сети. В частности, web-сервер не должен иметь доступа к информации, расположенной на клиентском компьютере, и тем более эта информация не должна передаваться обратно на сервер. Если бы у серверов имелась такая возможность, пользователи весьма неохотно использовали бы незнакомые web-сайты, что значительно затормозило бы развитие сети.

В 1994 г. компания Sun разработала браузер HotJava, чтобы продемонстрировать возможности содержащегося в нем виртуального компьютера Java. И наконец, 23 мая 1995 г. Марк Андерсен (Marc Anderssen), один из основателей компании Netscape Communication, которая в то время контролировала 70 % рынка в области web-браузеров, объявил о включении виртуального компьютера Java в браузер Netscape. С этого момента язык Java приобрел значительную популярность. Хотя исходно он был предназначен для выполнения апплетов в web-браузере, область его применения расширилась до того, что он стал одним из наиболее распространенных языков. Он все чаще заменяет C и C++ в качестве первого языка программирования, изучаемого и используемого студентами.

Краткий обзор языка. Язык Java похож на C и C++. Их взаимосвязь становится понятной, если проследить историю создания этих языков. Язык C был разработан в 70-х гг. как язык создания операционных систем (в основном системы UNIX). Разработчикам языка хотелось, чтобы он обеспечивал возможность доступа к архитектуре базового аппаратного компьютера. Когда Страуструп (Stroustrup) разрабатывал язык C++, он позаимствовал концепцию класса из языка Simula и концепцию наследования из языка Smalltalk. Тем не менее язык C, ставший основой C++, не был при этом модифицирован и его полезные для системного программирования свойства были перенесены в C++. Когда программисты компании Sun разрабатывали Java, они сохранили основные синтаксические структуры, концепции классов и наследования, имеющиеся в C++, но отказались от некоторых необязательных свойств. В результате язык Java оказался более компактным, чем C++, но с более рациональными синтаксисом и семантикой.

По большей части Java напоминает C++ без тяжелого наследия C, которое обусловливает слабость многих программ на языке C++. Данные в Java строго типизированы; в частности, целочисленный, логический и символьный типы принадлежат к различным типам данных. Также отдельным типом данных являются массивы, а строковый тип данных в Java — это не просто массив символов. Все эти свойства позволяют компилятору Java более полно отслеживать ошибки в программах.

Количество способов выполнения некоторых действий уменьшилось. Например, вызов метода — единственный способ подключения подпрограмм. Поскольку

все объекты являются экземплярами каких-либо классов, то отпадает необходимость в специальных вызовах функций или процедур. Структурные объекты struct также не нужны, так как тот же эффект достигается при помощи переменных экземпляра, или полей, в определении классов.

В языке Java неявным образом присутствуют указатели, но соответствующий тип данных отсутствует. Это означает, что от пользователя скрыты все проблемы с фрагментацией памяти, ссылками на несуществующие объекты и другие неприятности, связанные с указателями. Распределение памяти для указателей происходит неявным образом во время создания объектов класса и выполнения операции new.

Для создания программы на языке Java в первую очередь создается файл имя_файла.java. Имя файла должно совпадать с именем класса, создаваемого в данной программе. Когда программа написана, вызывается компилятор Java. В результате компиляции получается файл с названием имя_файла.class, содержащий байт-коды. Этот файл можно выполнить с помощью интерпретатора Java (виртуальная машина Java).

Можно сказать, что язык Java является достаточно простым языком: он имеет ту же ясную структуру, что и C++, и при этом лишен той неуклюжести, которая свойственна С. Тем не менее эффективность выполнения программ на Java несколько ниже, так как многие из структур языка нуждаются в проверке во время выполнения программы. К тому же, поскольку программы на Java чаще всего интерпретируются как апплеты в виртуальной машине Java, скорость выполнения программ на Java несколько ниже, чем скорость выполнения программ, написанных на компилируемых языках. Но при оценке Java надо помнить об основной области применения этого языка: поскольку он создавался для web-браузеров, потери в скорости выполнения фактически не влияют на результат. Причина этого в том, что скорость передачи информации по сети и скорость обработки и отображения информации на мониторе компьютера все равно ниже скорости работы компьютера пользователя. Поэтому большая часть времени тратится на ожидание очередной порции информации, запрошенной на сервере.

2.3. Рекомендуемая литература

В данном разделе не обсуждалась альтернативная архитектура компьютера, основанная на принципах, радикально отличающихся от принципов организации компьютера фон Неймана. Обзорную информацию по этому вопросу вы найдете в главе 11. Критика архитектуры фон Неймана и альтернативные предложения представлены в лекции Бэкуса [15], прочитанной при вручении ему премии Тьюринга. В настоящее время все большее значение приобретает параллельная архитектура и объединение большого количества компьютеров, работающих параллельно, в единую систему, что позволяет увеличить эффективность. Об экспериментальных исследованиях в области компьютерной архитектуры можно прочитать в январском выпуске IEEE Computer за 1991 г., а также в [77].

2.4. Задачи и упражнения

1. Проанализируйте реализацию какого-либо известного вам языка программирования. Что является исполняемой формой программы (иначе говоря, что представляет собой результат работы транслятора)? Какие виды трансляций происходят при преобразовании различных выражений и операторов в их исполняемую форму? Какая программная интерпретация необходима при выполнении программы? Является ли интерпретатор программно-моделируемым? Какие из примитивных операций требуют программной интерпретации?
2. Исследуйте свой собственный компьютер: какова структура виртуального компьютера, определяемого операционной системой? Чем этот компьютер отличается от аппаратного? Ограничивает ли операционная система использование аппаратных средств (например, существуют ли какие-либо аппаратные инструкции, которые становятся недоступными для использования в пользовательских программах, выполняемых определяемым операционной системой виртуальным компьютером)? Какие новые возможности, требующие сложного программного моделирования при реализации в базовом аппаратном компьютере, непосредственно обеспечиваются виртуальным компьютером, определяемым операционной системой (например, ввод-вывод)?
3. Операционная система позволяет программисту использовать виртуальный компьютер, который может сильно отличаться от базового аппаратного компьютера. Основные преимущества этого виртуального компьютера таковы. Пользователь взаимодействует с более простым компьютером, чем базовый (в частности, возможности ввода-вывода виртуального компьютера проще и мощнее). Он также защищает компьютерную систему от пользователя: так как каждый пользователь может быть изолирован в отдельном виртуальном компьютере, то любые его ошибки затронут только один этот виртуальный компьютер и не повлияют на работу всей компьютерной системы и других пользователей. Кроме того, виртуальный компьютер позволяет операционной системе более эффективно распределять системные ресурсы между пользователями. Проанализируйте вашу локальную операционную систему и виртуальный компьютер, предоставляемый ею программисту. Насколько хорошо операционная система вашего компьютера обеспечивает достижение перечисленных выше целей?
4. Язык программирования BASIC часто реализуется посредством полного программного моделирования виртуального компьютера языка BASIC. Если допустить, что моделирующие этот виртуальный компьютер программы написаны на языке FORTRAN (что вполне возможно), то в схеме, аналогичной той, что приведена на рис. 2.4, добавится еще один уровень. Изобразите этот дополнительный уровень. Если вы знакомы с обоими этими языками, подумайте, какие части виртуального компьютера языка BASIC могут быть смоделированы при помощи программ на языке FORTRAN. В каком месте упомянутой схемы расположится *транслятор*, который преобразует программы на языке BASIC в их выполняемую форму?

5. Напишите какой-либо оператор на знакомом вам языке. Для каждого синтаксического компонента (имена переменных, символы операций и т. д.) перечислите все возможные связывания, необходимые для полного определения семантики данного оператора при его выполнении. Для каждого связывания определите время связывания, используемое в этом языке.

6. Выполните задание 5, но уже не для оператора, а для *объявления*. Про объявления говорят, что они *обрабатываются* (elaborated), а не выполняются. Перечислите, какие связывания должны произойти для полной обработки этого объявления, и укажите времена этих связываний.

7. Посмотрите, какие языки обсуждаются в приложении. Какие из них, по вашему мнению, не войдут в следующее издание этой книги? Какие будут наиболее популярными в течение следующих десяти лет? Рассмотрите времена связываний различных объектов, виртуальный компьютер, необходимый для выполнения программ, и другие атрибуты языка, влияющие на эффективность его использования.

Глава 3. Вопросы трансляции языка

На раннем этапе разработки языков программирования (в 60-х гг., когда создавались такие языки, как FORTRAN, ALGOL, COBOL и LISP) считалось, что для написания программ достаточно соблюдения формальных правил синтаксиса. Была разработана концепция контекстно-свободной грамматики, или НФБ-грамматики (от «нормальная форма Бэкуса»), которая обсуждается в этой главе. Эта концепция с успехом применялась для спецификации синтаксиса языка, и до сих пор она является основным способом описания компонентов программы. Но со временем стало ясно, что одного синтаксиса недостаточно для решения всех вопросов, связанных с разработкой языков программирования. В главе 4 вы найдете краткое введение в семантику языка программирования — совокупность правил, определяющих смысл как языковых конструкций, так и программ в целом.

3.1. Синтаксис языка программирования

Синтаксис, определяемый как «система языковых категорий, относящихся к соединениям слов и строению предложений»[1], в языках программирования описывает последовательность символов, которая составляет синтаксически правильную программу. Например, в языке C оператор $X = Y + Z$ представляет собой правильную последовательность символов, а выражение $XY+-$ не является правильной последовательностью.

Синтаксис предоставляет важную информацию, необходимую как для понимания программы, так и для ее трансляции в объектную программу. Например, почти все читатели данной книги согласятся, что значение выражения $2 + 3 \times 4$ равно 14, а не 20. То есть это выражение интерпретируется как $2 + (3 \times 4)$, а не как $(2 + 3) \times 4$. При желании мы можем выбрать любую интерпретацию путем задания соответствующих синтаксических правил и таким образом дать транслятору

---

[1] *Ожегов С. И., Шведова Н. Ю.* Толковый словарь русского языка / РАН; Российский фонд культуры. 2-е изд., испр. и доп. М.: АЗЪ, 1995. 928 с. В оригинале здесь цитата и ссылка на: Webster's New World Dictionary, Fawcett, 1979. Мы сочли уместным в русском переводе привести цитату из толкового словаря русского языка. — *Примеч. науч. ред.*
Перевод определения из Webster's New World Dictionary примерно таков: «Структура предложения, состоящего из слов, выявляющая их взаимосвязь». — *Примеч. лит. ред.*

указание генерировать правильную последовательность операций для вычисления значения этого выражения.

Как и в случае с неоднозначным английским предложением «They are flying planes»[1], одного синтаксиса языка недостаточно для однозначной спецификации структуры оператора. Например, в операторе $X = 2.45 + 3.67$ синтаксис не скажет нам, была ли объявлена переменная X в программе, а если и была, то является ли она переменной вещественного типа. Результат выполнения этого оператора может быть $X = 5$, $X = 6$ или $X = 6.12$ в зависимости от того, как были определены тип переменной X и операция сложения «+». Если переменная X была определена как целая переменная, а операция «+» — как целочисленное сложение, то результат будет $X = 5$. Если X — целая переменная, а «+» обозначает сложение вещественных чисел, то в итоге получится $X = 6$. Если же и X, и «+» определены как относящиеся к вещественному типу, то результат будет $X = 6.12$. Таким образом, для полного описания языка программирования недостаточно только определения синтаксических структур. В языке имеются и другие атрибуты, известные под общим названием *семантика*, которые не всегда определяются синтаксическими правилами, но оказывают влияние на переменную. Сюда относятся использование объявлений, операций, управление последовательностью действий и среда ссылок.

Хотя описания синтаксиса недостаточно для полного понимания языка программирования, все же синтаксис является важным его атрибутом. По большей части описание синтаксиса является решенной задачей. Когда чуть позже мы будем в этой же главе обсуждать разработку транслятора, то увидим, что первая фаза — фаза синтаксического разбора исходной программы — представляет собой чисто механический процесс. Такие инструменты, как YACC (Yet Another Compiler Compiler) автоматически создают синтаксическое описание исходной программы. Использование семантики для генерирования эффективной объектной программы из заданной исходной программы все еще требует большого мастерства в сочетании с применением некоторых формальных методов.

3.1.1. Общие синтаксические критерии

Основным назначением синтаксиса языка программирования является обеспечение системы обозначений для обмена информацией между программистом и процессором языка программирования. Выбор конкретных синтаксических структур, тем не менее, только в небольшой степени определяется необходимостью передачи конкретной информации. Например, тот факт, что значение некоторой переменной принадлежит к типу вещественных чисел, можно выразить дюжиной различных способов — посредством явного описания в языке C или неявного соглашения о выборе имен в языке FORTRAN и т. д. При разработке деталей синтаксиса по

---

[1] Это предложение можно перевести двояко: «Они летят самолетами» и «Они являются летящими самолетами». — *Примеч. пер.*

Эта фраза имеет двоякий смысл лишь ввиду игнорирования артиклей и предлогов, что не свойственно уважающему родной язык англичанину. Американцы, не столь щепетильные в вопросах языка, могут считать эту фразу имеющей право на существование, тогда как англичане посчитают ее заведомо ошибочной: должно быть либо «They are flying by planes», либо «They are the flying planes». — *Примеч. лит. ред.*

большей части исходят из второстепенных соображений (таких, например, как простота чтения программы), которые не связаны напрямую с основной задачей передачи информации процессору языка.

Второстепенных критериев довольно много, но их можно разбить на четыре основные группы: легкость чтения, написания и трансляции программы, а также однозначность. Мы рассмотрим некоторые подходы к разработке синтаксических структур языка, которые удовлетворяют указанным (часто противоречащим друг другу) целям.

Легкость чтения. Программа легка для чтения, если структура ее алгоритма и представленные в ней данные становятся очевидны при просмотре ее текста. Легкую для чтения программу называют также *самодокументируемой*, так как для ее понимания не требуется никакой дополнительной документации (на практике эта цель редко достигается). Легкости чтения способствуют такие качества языка, как естественные форматы операторов, структурированные операторы, свободное использование ключевых и необязательных слов, возможность встраивания в текст программы комментариев, неограниченность длины идентификаторов, мнемонические символы операций, запись программы в свободном формате и полный набор объявлений используемых данных. Безусловно, легкость чтения программы не может гарантироваться только лишь идеологией языка, так как самая лучшая идеология не может стать панацеей от плохого программирования. Однако синтаксические конструкции языка могут быть таковы, что даже программист, имеющий самые лучшие намерения, будет писать на этом языке неудобочитаемые программы (примером может служить APL). Наиболее ярко стремление к удобочитаемости выражено в COBOL, но, к сожалению, лишь за счет удобства записи и трансляции.

Улучшает чтение программы такой синтаксис языка, в котором синтаксические различия отражают лежащие в его основе семантические особенности. Таким образом, программные конструкции, имеющие схожее назначение, должны выглядеть одинаково, а те конструкции, которые выполняют различные действия, и выглядеть должны по-разному. В целом, чем больше разнообразие синтаксических конструкций, тем проще отразить в структуре программы различные семантические структуры, соответствующие этим конструкциям.

Программы, написанные на языках, в которых сравнительно мало различных синтаксических конструкций, обычно труднее читать. Например, в APL и SNOBOL4 предусмотрен только один формат операторов. Различия между оператором присваивания, вызовом подпрограммы, оператором goto, выходом из подпрограммы, многовариантным условным ветвлением и многими другими распространенными программными структурами синтаксически отражены только различиями в одном или нескольких символах операторов внутри сложного выражения. Часто даже для определения крупной структуры управления программы требуется детальное ее изучение. Более того, такая незначительная ошибка, как, например, неправильное написание одного символа в операторе, может совершенно изменить его смысл, причем сам оператор может быть синтаксически правильным. В LISP к подобным последствиям приводит несоответствие между числом открывающих и закрывающих скобок. Одно из расширений этого языка предназначено специально для того, чтобы избежать этой проблемы.

Легкость написания. Синтаксические особенности, способствующие легкости написания программы, часто вступают в противоречие с теми, которые делают ее удобочитаемой. Легкость написания программ обеспечивается использованием кратких и однородных синтаксических структур, в то время как для удобства чтения программы требуется разнообразие конструкций. Язык С, к сожалению, является примером языка, программы на котором можно записать очень лаконично, но читать их будет очень сложно, хотя этот язык и имеет полный набор полезных свойств.

Неявные синтаксические соглашения, которые позволяют не определять операции и типы переменных, упрощают написание программы и укорачивают запись, но усложняют чтение программы. Но есть и такие средства, которые способствуют достижению обеих целей: использование структурированных операторов, простых естественных форматов операторов, мнемонических символов операций и неограниченных по длине идентификаторов. Эти средства упрощают написание программы, так как алгоритм задачи и данные оказываются представленными в программе в их естественной структуре.

Синтаксис называют *избыточным*, если одну и ту же информацию можно передать более чем одним способом. Некоторая избыточность способствует легкости чтения программы и также позволяет проверять ошибки во время трансляции. Недостатком же избыточности синтаксиса является то, что он делает программы более многословными и, следовательно, затрудняет их написание. Большинство используемых по умолчанию правил для определения смысла языковых конструкций направлены на уменьшение избыточности синтаксиса: если из контекста программы однозначно следует некоторый вывод о какой-либо конструкции, то от явной формулировки этого вывода можно отказаться. Например, в языке ML вместо того, чтобы явно описывать тип данных для каждого параметра функции, используется принцип индукции для определения их типов данных. Но если, однако, в написании такой функции произошла ошибка, то транслятор будет не в состоянии ее обнаружить. Подобный эффект маскирования ошибок в программах является причиной того, что языки, полностью свободные от избыточности, зачастую трудно использовать.

Легкость верификации. С легкостью чтения и написания программ связана концепция корректности программ, также называемая ***верификацией программ***. Многолетний опыт написания программ позволяет утверждать, что разобраться в любом отдельно взятом операторе относительно просто, но сам процесс написания правильной программы чрезвычайно сложен. Следовательно, необходимы методики, позволяющие математически строго доказывать корректность написанных программ. Мы обсудим эту тему в главе 4.

Легкость трансляции. Третья цель, которая противоречит первым двум, — это достижение легкости трансляции программы в исполняемую форму. Легкость чтения и записи — это критерии, определяемые нуждами программиста. Легкость трансляции относится к нуждам транслятора, который обрабатывает написанную программу. Ключевым моментом для упрощения трансляции является регулярность структуры программы. LISP является примером языка, программы которого не являются слишком легкими ни для чтения, ни для записи, но при этом исключительно просты для трансляции. Благодаря регулярности синтаксиса вся

синтаксическая структура любой LISP-программы может быть описана несколькими простыми правилами. Сложность трансляции программы возрастает по мере увеличения количества специальных синтаксических конструкций. Например, очень сложно транслировать программы, написанные на языке COBOL, из-за большого количества допустимых форм операторов и объявлений, хотя этот язык нельзя назвать семантически сложным.

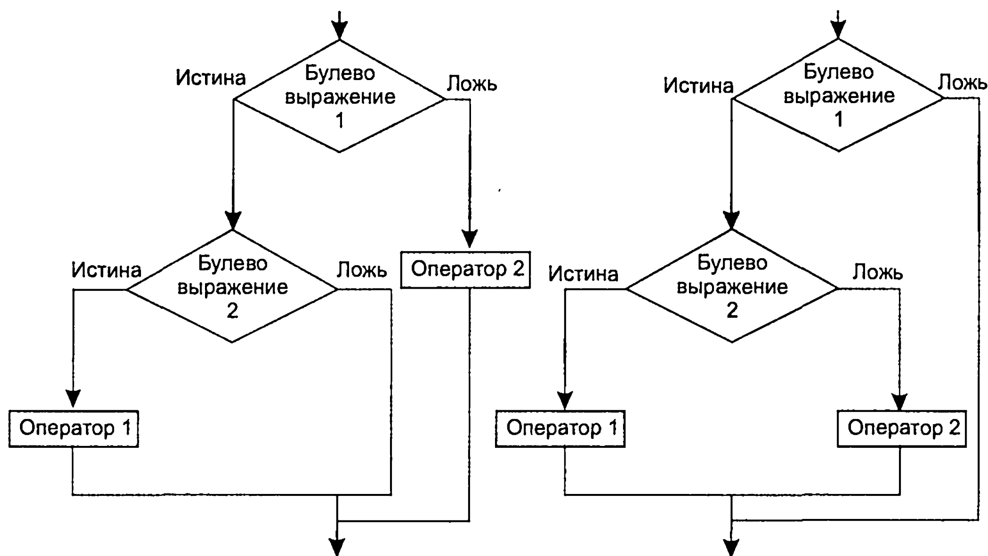


Рис. 3.1. Две интерпретации условного оператора

Отсутствие неоднозначности. Неоднозначность является центральной проблемой при разработке любого языка. Определение языка в идеале должно обеспечивать однозначность любой синтаксической конструкции, которую программист может написать. Неоднозначная конструкция допускает два и более толкований. Проблема неоднозначности обычно возникает не в структуре отдельно взятого элемента программы, а при взаимодействии различных структур. Например, и Pascal, и ALGOL допускают две различные формы условного оператора:

```
if булево_выражение then оператор1 else оператор2
if булево_выражение then оператор1
```

При интерпретации каждого из этих операторов их смысл ясно определен. Однако если эти две формы объединяются вместе и в качестве первого оператора берется вторая форма условного оператора, то получается следующая структура, которая называется условным оператором с *повисшим* else:

```
if булево_выражение1 then if булево_выражение2 then оператор1 else оператор2
```

Эта структура является неоднозначной, так как неясно, какая из представленных на рис. 3.1 последовательностей вычислений имеется в виду. Другой пример можно позаимствовать из языка FORTRAN. Ссылка на A(I.J) может означать как ссылку на элемент двухмерного массива A, так и обращение к подпрограмме-функции A с двумя параметрами I и J, поскольку в FORTRAN для вызова функции

и ссылки на элемент массива используется одна и та же синтаксическая конструкция. Подобные примеры неоднозначности можно найти в любом языке программирования.

Проблемы с упомянутыми выше неоднозначными конструкциями в FORTRAN и ALGOL фактически были решены в обоих этих языках. В ALGOL неоднозначность конструкции, состоящей из двух или более условных операторов, была устранена путем изменения синтаксиса языка — добавилась пара разделителей begin ... end, отделяющая вложенный условный оператор от внешних конструкций. Таким образом, естественная, но неоднозначная конструкция, состоящая из двух условных операторов, была заменена двумя менее естественными, зато однозначными конструкциями, из которых можно выбрать ту или иную в зависимости от требуемой интерпретации:

```
if булево_выражение1 then begin if булево_выражение2 then оператор1 end else оператор2
if булево_выражение1 then begin if булево_выражение2 then оператор1 else оператор2 end
```

Более простое решение используется в языке Ada: каждый условный оператор if должен заканчиваться разделителем end if. В C и Pascal используется другой метод устранения неоднозначности. Для неоднозначной конструкции избирается произвольное толкование — в данном случае последнее else согласуется с ближайшим к нему then, так что в результате получается комбинированный оператор, смысл которого совпадает со второй из приведенных выше конструкций языка ALGOL. Неоднозначность ссылок на функции и элементы массивов в языке FORTRAN разрешена следующим образом: конструкция A(I,J) рассматривается как вызов функции, если не был объявлен массив A. Поскольку каждый массив должен быть описан до того, как он будет использован в программе, для транслятора не составит труда проверить, был ли объявлен массив A, на который имеется ссылка. Если объявление массива не обнаружено, то транслятор предполагает, что в данном случае имеется в виду обращение к внешней функции A. Это предположение может быть проверено только во время загрузки, когда все внешние функции (в том числе библиотечные) будут собраны в окончательную исполняемую программу. Если теперь загрузчик не обнаружит функцию A, то появится сообщение загрузчика об ошибке. В Pascal используется другая техника для устранения подобной неоднозначности. Вводится синтаксическое отличие: в квадратные скобки заключается список индексов при ссылке на элемент массива (например, A[I,J]), а в круглые — список параметров при обращении к функции (например, A(I,J)).

3.1.2. Синтаксические элементы языка

Общий стиль синтаксиса языка определяется выбором основных синтаксических элементов. Мы кратко рассмотрим наиболее характерные из них.

Набор символов. Определение набора используемых символов — это одна из первоочередных задач, которую необходимо решить при разработке синтаксиса языка программирования. Существует несколько широко используемых наборов, например, таких как ASCII[1], в каждом из которых помимо основных букв и цифр

---

[1] American Standard Code for Information Interchange (ASCII) — американский стандартный код обмена информацией. — *Примеч. науч. ред.*

содержится свой ряд специальных символов. Обычно выбирается один из таких стандартных наборов, хотя иногда используется какой-либо специальный нестандартный набор, как, например, это сделано в APL. Выбор допустимого множества символов важен для определения типа устройств ввода-вывода, которые могут быть использованы в реализации языка. Например, множество основных символов языка С доступно почти на всех устройствах ввода-вывода, в то время как множество символов языка APL невозможно непосредственно использовать в большинстве устройств ввода-вывода.

Использование 8-битных байтов для представления символов (один символ — один байт) в начале 60-х гг., когда произошел переход от 6-битового к 8-битовому представлению символов, казалось весьма разумным выбором. Таблица из 256 символов казалась более чем достаточной для того, чтобы представить в верхнем и нижнем регистрах 52 буквы латинского алфавита, десять цифр и несколько знаков препинания. Но в наше время компьютерная индустрия стала гораздо более интернациональной. Не все (на самом деле очень немногие) страны используют один и тот же набор из 26 букв латинского алфавита. В испанском алфавите присутствует тильда (~), во французском — диакритические знаки (´, `, ^), а в других языках — и некоторые дополнительные специальные символы (например, å, β, ö и др.). Кроме того, имеются такие языки, как греческий, иврит и арабский, где набор символов радикально отличается от существующего в европейских языках. Представление китайского или японского языка, в которых каждый иероглиф соответствует слову или фразе, требует набора, состоящего из примерно 10 000 символов. Во всех этих случаях, очевидно, недостаточно однобайтного представления символов. Разработчики языков сейчас все больше склоняются к двухбайтному представлению символов (в котором каждый символ представляется шестнадцатью битами, что расширяет количество допустимых символов до 65 536)[1].

Идентификаторы. Для идентификаторов широко используется следующий синтаксис — строки, состоящие из букв и цифр, начинающиеся с буквы. Различия между правилами составления идентификаторов в разных языках сводятся к включению в допустимый набор букв и цифр некоторых специальных символов, например точки или дефиса (что облегчает восприятие при чтении программы), и тому или иному ограничению длины идентификатора. Ограничение длины (например, в ранней версии языка BASIC эта длина не превышала одной буквы и последующей цифры) вынуждает программиста использовать идентификаторы, лишенные выраженного мнемонического значения, что приводит к значительной потере удобочитаемости программы.

Символы операций. В большинстве языков символы «+» и «-» используются для обозначения двух основных арифметических операций, но на этом сходство в использовании символов для одинаковых операций в разных языках и заканчивается. Все простейшие операции могут быть представлены исключительно специальными символами, как это сделано в APL; можно, наоборот, для представления всех этих операций использовать идентификаторы, как, например, это сделано в язы-

---

[1] В настоящее время наиболее употребительные языки, особенно те, которые используются для создания Интернет-приложений, поддерживают, а некоторые, например Java и Perl, используют как внутреннее представление символов двухбайтную кодировку Unicode (UTF-8). — *Примеч. науч. ред.*

ке LISP: PLUS для сложения, TIMES для умножения и т. д. В большинстве языков применяется некая комбинация упомянутых подходов: для обозначения некоторых операций используются символы, для других — идентификаторы, и также часто используются строки символов, которые нельзя отнести к какой-либо из предыдущих двух категорий (например, в языке FORTRAN последовательности .EQ. и ** используются для обозначения соответственно операций сравнения на равенство и возведения в степень).

Ключевые и зарезервированные слова. *Ключевое слово* — это идентификатор, используемый в качестве фиксированной части синтаксиса какого-либо оператора (например, слово `if`, служащее началом условного оператора в языке С, или слово `for`, начинающее оператор цикла). Ключевое слово является *зарезервированным*, если синтаксис запрещает его использование в качестве идентификатора, определяемого программистом. Зарезервированные слова имеются в большинстве используемых в настоящее время языков программирования, улучшая, таким образом, возможности транслятора по обнаружению синтаксических ошибок. Большинство операторов начинаются с зарезервированного слова, указывающего на тип оператора: `READ`, `IF`, `WHILE` и т. д.

Использование зарезервированных слов облегчает синтаксический анализ во время трансляции. В языке FORTRAN, например, синтаксический анализ осложняется из-за того, что оператор, начинающийся со слов `DO` или `IF`, может не быть оператором цикла или условным оператором. Поскольку `DO` и `IF` не являются зарезервированными словами, программист имеет полное право использовать их в качестве имен переменных. В языке COBOL ситуация в некотором смысле противоположная — зарезервированных слов так много, что все их почти невозможно запомнить; в результате человек может случайно употребить какое-либо из этих слов в качестве имени переменной. Основная трудность с зарезервированными словами, однако, проявляется тогда, когда возникает необходимость в расширении языка включением новых операторов, использующих новые зарезервированные слова (как происходило, например, при пересмотре языка COBOL). Добавление нового зарезервированного слова означает, что любая программа, написанная ранее на этом языке, становится синтаксически неправильной, если в ней используется в качестве имени переменной (или в каком-либо ином качестве) введенное в очередной стандарт новое зарезервированное слово, хотя сама программа при этом никак не была изменена.

Необязательные слова. Необязательные слова — это слова, которые вставляются в операторы для удобства чтения. Множество таких слов имеется в языке COBOL. Например, оператор `goto` в COBOL, записанный в форме `GO TO метка`, представляет пример оператора, в котором ключевое слово `GO` является обязательным, а `TO` — необязательным. Оно не несет никакой информации, а просто облегчает чтение программы.

Комментарии. Включение комментариев в программы является важной частью ее документирования. Язык позволяет включать комментарии несколькими способами:

1) в специальной строке комментария (например, в программе на BASIC комментарии располагаются в строке, начинающейся с ключевого слова `REM`);

2) в произвольном месте программы, при этом текст комментария отделен от основного текста программы специальными маркерами (в языке С, например, таковыми служат последовательности символов /* и */);
3) начало комментария может располагаться в произвольном месте, но его конец должен совпадать с концом строки; так, например, в языке Ada начало комментария обозначается последовательностью дефисов - -, в FORTRAN 90 — символом !, а в C++ — символами //.

Недостаток второго из этих способов заключается в том, что если пропущен символ, обозначающий конец комментария, весь следующий далее текст программы (до конца следующего комментария) будет воспринят как комментарий. Следовательно, он не будет обработан транслятором и не выполнится, хотя эта часть текста может выглядеть совершенно правильно при чтении программы.

Пробелы. В разных языках правила использования пробелов варьируются в широких пределах. Например, в языке С пробелы не являются значащими символами, за исключением случаев их использования в строковых литералах. В других языках пробелы используются в качестве разделителей, поэтому они играют важную синтаксическую роль. В SNOBOL4 элементарная операция конкатенации представлена пробелом, который также используется как разделитель между элементами в операторах (что приводит к большой путанице). В С пробелы по большей части игнорируются, но не всегда. В ранних версиях С пара символов «=+» являлась единой операцией, а запись = + представляла собой последовательность двух операций. Для предотвращения подобных ошибок в определение С были внесены некоторые изменения, и теперь символ «+=» является символом комбинированной операции, а использование + = приводит к синтаксической ошибке.

Разделители и скобки. *Разделитель* — это синтаксический элемент, функция которого заключается в обозначении начала или конца некоторой синтаксической конструкции, например оператора или выражения. Скобки — это парные ограничители, например *круглые скобки* или пары типа begin ... end. Разделители могут использоваться просто для облегчения чтения программы или для упрощения синтаксического анализа, но чаще они применяются для решения серьезной задачи — устранения неоднозначности путем явного определения границ конкретной синтаксической конструкции.

Свободный и фиксированный форматы. Синтаксис с фиксированным форматом записи операторов языка в программе — это пережиток эпохи перфокарт. Синтаксис называется синтаксисом со свободным форматом записи операторов, если он допускает запись операторов программы в произвольном месте строки ввода безотносительно к его положению в строке или разрывам между строками. Синтаксис с фиксированным форматом записи операторов использует определенные области строки ввода для передачи определенной информации. Такой синтаксис, в котором каждый элемент оператора должен располагаться в определенном месте строки ввода, характерен для языков ассемблера. В настоящее время синтаксис с фиксированным форматом записи операторов встречается все реже, уступая синтаксису со свободным форматом, который теперь является нормой.

Выражения. Выражения — это функции, которые обрабатывают какие-либо данные в программе и возвращают некоторые значения. Выражения являются ос-

новными синтаксическими блоками, из которых строится оператор (а иногда и программа). В императивных языках (например, в языке C) выражения формируют основные операции, позволяющие изменять состояния компьютера при выполнении каждого оператора. В функциональных языках, таких как ML или LISP, выражения используются для управления последовательностью действий. Мы обсудим построение выражений более подробно в главе 8.

Операторы. Операторы являются самыми важными синтаксическими компонентами императивных языков — доминирующим в настоящее время классом языков. От синтаксиса операторов зависят регулярность языка, удобство чтения и записи программы. В некоторых языках принят единственный формат записи операторов, в других используются различные форматы для различных типов операторов. В первом случае обеспечивается регулярность языка, во втором — удобство чтения. В SNOBOL4 имеется только синтаксис одного основного оператора — оператора сопоставления с образцом и подстановки, все остальные типы операторов получаются из основного путем удаления тех или иных его элементов. Но в большинстве языков наблюдается другая крайность: каждый тип операторов имеет свою синтаксическую структуру. В этом отношении самым ярким представителем является COBOL: каждый оператор этого языка имеет свою особую структуру, в которую входят ключевые и необязательные слова, альтернативные конструкции, необязательные элементы и т. д. Преимущество использования разнообразных синтаксических структур, конечно, заключается в том, что операции могут быть выражены естественным образом.

Более существенное различие в структурах операторов затрагивает различие между *структурированными* (или *вложенными*) и *простыми* операторами. В простом операторе, очевидно, не содержится других встроенных операторов. В языках APL и SNOBOL4 допускаются только простые операторы. В структурированном операторе могут содержаться другие встроенные операторы. Преимущества структурированных операторов обсуждаются более подробно в главе 8.

3.1.3. Общая структура программы-подпрограммы

Общая синтаксическая организация основной программы и определений подпрограмм варьируется в таких же широких пределах, как и другие аспекты синтаксиса языка.

Отдельные определения подпрограмм. Язык C является примером такой организации языка, при которой каждое отдельное определение подпрограммы рассматривается как отдельный синтаксический компонент. Каждая подпрограмма компилируется отдельно, и затем все они связываются в единую программу во время загрузки. Объектная ориентация требует, чтобы информация передавалась от одного к другому независимо откомпилированному компоненту программы. Наследование определений классов требует, чтобы компилятор обработал некоторые из этих отдельных подпрограмм, прежде чем программа будет загружена и начнет выполняться.

Отдельные определения данных. Альтернативной моделью является объединение в одну группу всех операций, которые взаимодействуют с конкретным объектом данных. Например, подпрограмма может состоять из всех операций, связан-

ных с обработкой данных определенного формата, имеющихся в программе: операций создания записи данных (data record), операций печати записи данных и операций вычисления с записью данных. Это общий подход для языков, в которых имеется механизм *класса*, например Java, C++ и Smalltalk.

Вложенные определения подпрограмм. Вложенное определение подпрограмм играло существенную роль для создания модульных программ на раннем этапе развития таких языков программирования, как ALGOL, FORTRAN и Pascal. Но с появлением объектно-ориентированных языков типа C++ и Java эта концепция утратила свое значение. Pascal является примером языка со вложенной структурой программ, при которой определения подпрограмм играют роль объявлений в основной программе. Сами эти подпрограммы, в свою очередь, могут содержать другие вложенные определения подпрограмм, и так до любого уровня вложенности. Вложенные определения подпрограмм обеспечивают для этих подпрограмм среду нелокальных ссылок, которая определяется во время компиляции и допускает статическую проверку типов и компиляцию эффективного выполняемого кода для подпрограмм, содержащих нелокальные ссылки.

Независимые определения интерфейса. Структура языка FORTRAN легко позволяет компилировать отдельные подпрограммы, но недостатком такого подхода является то, что одни и те же данные, используемые в разных подпрограммах, могут по-разному быть объявлены в них. Следствием этого является неспособность компилятора обнаружить подобные несоответствия во время компиляции. Pascal, напротив, позволяет осуществлять такую проверку, так как компилятор языка Pascal имеет доступ ко всем определениям данных, но недостатком такого подхода является необходимость перекомпиляции всей программы при внесении в нее любого изменения (что, конечно, в случае длинных программ сильно замедляет процесс разработки). В языках Ada, ML и C для улучшения качества работы компилятора используются возможности обеих методик.

В названных языках *реализация* программы состоит из нескольких подпрограмм, которые должны взаимодействовать друг с другом. Все эти компоненты (называемые *модулями*) связываются воедино, как в языке FORTRAN, и образуют выполняемую программу; и при изменении какого-либо модуля не приходится компилировать заново всю программу, но только один измененный модуль. Тем не менее данные, которые используются в нескольких подпрограммах модуля, должны иметь общие объявления, как в языке Pascal, чтобы обеспечить их эффективную проверку компилятором. Однако для передачи информации между независимо компилируемыми модулями нужны дополнительные данные. Для этих целей используется компонент *спецификации программы* (program-specification component). В языке C выбран подход, использующий включение в него некоторых файловых операций операционной системы, что позволяет включать в программу файлы, содержащие определения интерфейса. Файлы с расширением .h формируют компоненты спецификации программы, а файлы программ с расширением .c формируют компоненты реализации[1]. В языке Ada выбран другой подход: эти сред-

[1] В UNIX имеется популярная программа make, которая используется для определения того, какие из файлов с расширениями .c и .h были изменены. Это позволяет компилировать заново только те компоненты, которые были изменены.

ства встроены в язык непосредственно. Программы определены в компонентах, называемых *пакетами*, в которых содержатся либо спецификации определений интерфейса, либо реализация исходной программы (в *теле* пакета), использующей эти определения интерфейса.

Описания данных, отделенные от выполняемых операторов. Программа на языке COBOL являет собой пример программы, использующей компонентные структуры, характерные для раннего этапа развития языков программирования. В ней описание данных и выполняемые операторы для всех подпрограмм разделены на два отдельных блока: *раздел данных* и *раздел процедур*. Третий раздел называется *разделом оборудования* и содержит описания, относящиеся к внешней операционной среде. Раздел процедур программы организован в виде блоков, каждый из которых соответствует телу какой-либо подпрограммы, но все данные являются глобальными для всех подпрограмм, и ничто не соответствует обычным локальным данным подпрограмм. Преимущество такого централизованного раздела данных, содержащего объявления всех данных, заключается в том, что появляется логическая независимость форматов данных и алгоритмов в разделе процедур. Незначительные изменения в структуре данных можно осуществить модификацией раздела данных, не затрагивая раздел процедур. Помимо того, удобнее сосредоточить все описания данных в одном месте, а не рассеивать их по многочисленным подпрограммам.

Неотделенные определения подпрограмм. SNOBOL4 представляет собой пример языка, совершенно противоположного языкам с четкой программной организацией — она в нем полностью отсутствует. В этом языке не существует никаких синтаксических различий между операторами основной программы и операторами подпрограмм. Независимо от количества имеющихся подпрограмм вся программа синтаксически является просто набором операторов. Точки начала и конца подпрограмм синтаксически не определены. Программа просто выполняется, обращение к функции начинает выполнение новой подпрограммы, а обращение к функции RETURN завершает ее выполнение. Поведение программы является полностью динамическим. Фактически любой оператор может быть частью основной программы и одновременно — частью любого числа подпрограмм в том смысле, что он может быть выполнен в какой-то момент как часть основной программы, а позже — как часть выполняемой подпрограммы. Такая довольно хаотичная организация программы имеет смысл, только если ее трансляция происходит во время выполнения и выполнение новых операторов и подпрограмм осуществляется с помощью сравнительно простых механизмов.

Язык BASIC [68] является примером еще одного языка, программы которого выполняются вопреки некоторым целям, преследуемым при разработке языков программирования. Он был разработан в Дартмутском колледже (Dartmouth College) в 60-х гг. Основной целью разработчиков было создать язык, который могли бы с легкостью использовать люди, далекие от программирования и точных наук. Такой язык, безусловно, должен был обладать чрезвычайно простым синтаксисом. Хотя длинные программы на языке BASIC довольно трудны для понимания, этот язык чрезвычайно эффективен для написания коротких «одноразовых» программ, которые требуется выполнить всего несколько раз, после чего их можно и уничтожить. Во врезке «Обзор языка 3.1» приведено краткое описание

BASIC. Однако развитие этого языка продолжалось, и последующие его версии становились все более сложными, часто достигая структуры языков типа Pascal в отношении мощности, синтаксиса и сложности.

Обзор языка 3.1. BASIC

Характеристика. Язык с чрезвычайно простыми синтаксисом и семантикой: пронумерованные операторы, имена переменных, состоящие из одной буквы и цифры, простой оператор IF, цикл FOR и оператор GOSUB для вызова подпрограммы.

История. BASIC (Beginner's All-purpose Symbolic Instruction Code) был разработан Томасом Куртцем (Thomas Kurtz) и Джоном Кемени (John Kemeny), сотрудниками Дартмутского колледжа (Dartmouth College), в начале 60-х гг. Целью разработчиков было создать простой в использовании язык программирования, в особенности для студентов не технических специальностей. Для увеличения эффективности вычислений BASIC был реализован как интерактивный язык, причём это было сделано задолго до того, как режим разделения времени стал стандартом системной архитектуры.

BASIC также является ярким примером внутренне противоречивого языка программирования. Те, кто критикует этот язык, повторяют шутку, что BASIC — это оксиморон[1] из одного слова. Хотя синтаксис языка BASIC очень прост в изучении, однако при попытке написать программу длиной более одной страницы сложность соединения компонентов программы в единое целое делает получающийся код совершенно нечитаемым. Именно по этой причине в следующие версии BASIC были внесены изменения, связанные с допущением более длинных имен переменных, включением в язык имен подпрограмм и увеличением гибкости структур управления. Эти изменения привели к тому, что BASIC стал похож скорее на язык типа Pascal и FORTRAN, чем на свой исходный простой вариант образца 60-х гг.

Пример. Программа, вычисляющая сумму $1^2 + 2^2 + ... + 10^2$

```
100 REMARK S IS SUM: I IS INDEX
200 LET S=0
300 FOR I=1 TO 10
400 LET S = S +I*I
500 NEXT I
600 REMARK NEXT IS END OF LOOP
700 PRINT "SUM IS ", S
800 STOP
900 REMARK OTHER STATEMENTS: IF S>400 THEN 200 - (branch to 200)
1000 REMARK OTHER STATEMENTS: DIM A(20) - A array of 20
1100 REMARK OTHER STATEMENTS: GOSUB 100: RETURN - Subroutines
1200 REMARK OTHER STATEMENTS: READ A-Input
```

Ссылка. T. E. Kurtz, "BASIC", ACM History of Programming Languages Conference, Los Angeles (June 1978) (SIGPLAN Notices (13)8 [August 1978]),103–118.

[1] Оксиморон (оксюморон) — стилистический оборот, в котором сочетаются семантически контрастные слова, создающие неожиданное смысловое единство, например «живой труп», «убогая роскошь». — *Примеч. науч. ред.*

3.2. Этапы трансляции

Процесс трансляции исходной программы, представленной синтаксическими структурами, в ее исполняемую форму — это основной процесс при реализации любого языка программирования. Он может оказаться достаточно простым, как в случае языков Perl, LISP или Prolog, но чаще он довольно сложен. Большинство языков могли бы быть реализованы с весьма тривиальным процессом трансляции, если бы можно было согласиться с написанием программного интерпретатора и низкой скоростью выполнения получаемого кода. Однако эффективность выполнения программы настолько важна, что большая часть усилий направлена на транслирование программы в эффективно выполняемые структуры. Процесс трансляции заметно усложняется по мере того, как форма выполняемой программы все более отдаляется от структуры исходной программы. В предельном случае оптимизирующий компилятор для сложного языка, например языка Ada, может полностью изменить программные структуры, чтобы добиться более эффективного выполнения.

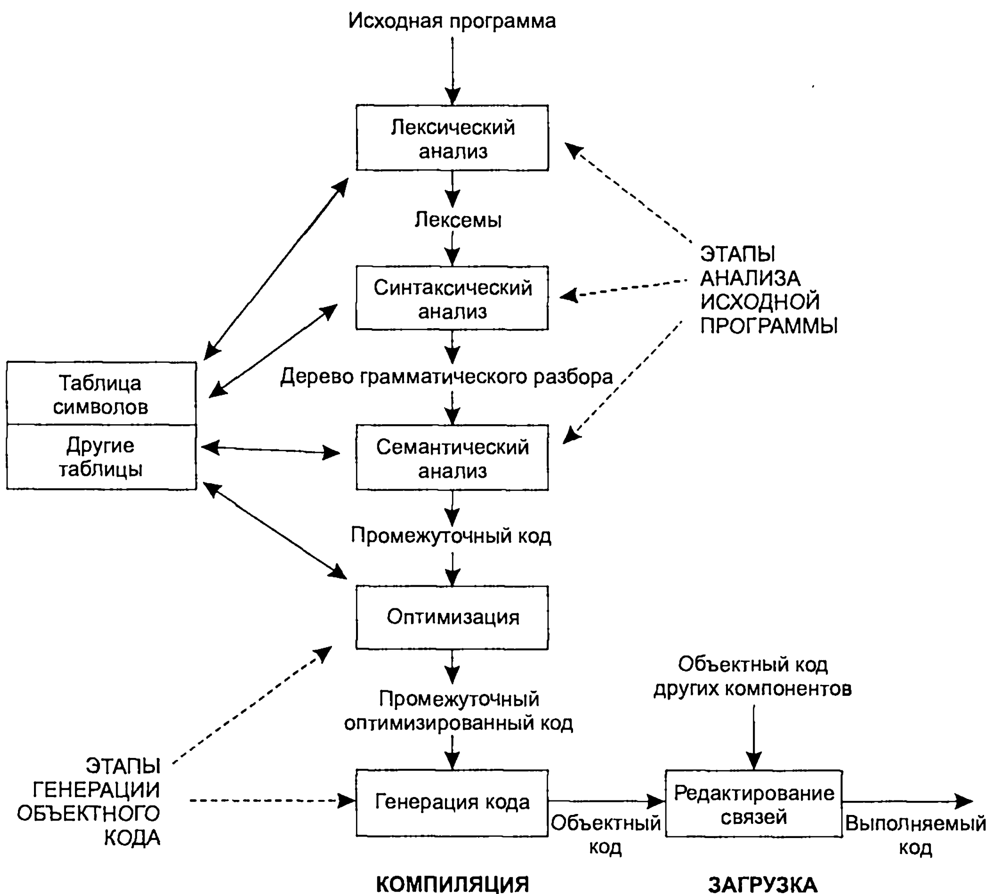


Рис. 3.2. Структура компилятора

Логически процесс трансляции можно разбить на две основные части: *анализ* исходной программы и *синтез* выполняемой объектной программы. В большинстве трансляторов эти логические стадии не имеют четкой границы; чаще они бывают настолько взаимосвязаны, что анализ и синтез чередуются — часто на уровне операторов. На рис. 3.2 представлена структура типичного компилятора.

Трансляторы можно грубо разделить на группы в зависимости от количества *проходов*, которые они делают по тексту исходной программы. Простой компилятор обычно использует два прохода. Во время первого прохода на основе анализа программы осуществляется ее декомпозиция на составляющие, а также получается необходимая для следующего прохода информация, например сведения об использовании имен переменных в программе. Во время второго прохода на основе полученной информации обычно генерируется объектная программа.

Если первоочередное значение имеет скорость компиляции (например, для компиляции учебных программ), при разработке транслятора можно применить стратегию единственного прохода. В таком случае программа сразу же после ее анализа преобразуется в объектный код. Архитектура языка Pascal разрабатывалась именно так, чтобы для него можно было разработать однопроходный компилятор. Но если скорость выполнения программы имеет большее значение, то следует разрабатывать компилятор, делающий три (или более) прохода. Во время первого прохода осуществляется анализ исходной программы, во втором проходе исходная программа переписывается в более эффективную форму с использованием различных алгоритмов оптимизации, а во время третьего прохода генерируется объектный код.

По мере того как расширяются наши познания в области технологии компиляции, взаимосвязь между количеством проходов и скоростью компиляции становится все менее очевидной. Более важным фактором оказывается сложность самого языка, а не количество проходов, совершаемое компилятором для анализа программы.

3.2.1. Анализ исходной программы

Для транслятора исходная программа представляет собой длинную однообразную цепочку, состоящую из тысяч или десятков тысяч символов. Разумеется, программисту текст программы представляется состоящим из подпрограмм, операторов, объявлений и т. п. Для транслятора такого разделения не существует. Транслятор производит анализ структуры программы, разбирая ее текст последовательно, символ за символом.

Лексический анализ, или сканирование. Начальный этап любой трансляции состоит в выделении в исходной программе элементарных составляющих: идентификаторов, разделителей, символов операций, чисел, ключевых и необязательных слов, пробелов, комментариев и т. д. Этот этап называется *лексическим анализом*, а те программные единицы, которые получаются в результате такого анализа, называются *лексическими единицами*, или *лексемами*. Обычно лексический анализатор (или сканер) — это программа ввода для транслятора, которая последовательно читает строки исходной программы, разбивает их на отдельные лексемы и передает эти лексемы на дальнейшие стадии трансляции, чтобы использовать их в анали-

зе более высокого уровня. Лексический анализатор должен идентифицировать тип каждой лексемы (число, ограничитель, идентификатор, оператор и т. д.) и пометить ее как принадлежащую соответствующему типу. Кроме того, часто делается перевод во внутреннее машинное представление для таких элементов, как числа (они преобразуются во внутреннюю двоичную форму записи с фиксированной или плавающей точкой) и идентификаторы (они заносятся в таблицу символов, и затем вместо строки символов используется адрес из этой таблицы). Формальной моделью, используемой для создания лексического анализатора, являются *конечные автоматы*. Их краткое описание вы найдете в разделе 3.3.2.

Хотя концепция лексического анализа достаточно проста, фактически этот этап трансляции занимает больше времени, чем любой другой. Частично это происходит просто потому, что транслятор вынужден сканировать и анализировать исходную программу символ за символом. Имеет значение и то, что на практике иногда бывает трудно определить границы между лексемами, не прибегая к помощи достаточно сложных контекстно-зависимых алгоритмов. Например, два оператора FORTRAN

```
DO 10 I = 1.5
```

и

```
DO 10 I = 1.5
```

состоят из совершенно различных синтаксических структур. Первый оператор — это оператор цикла DO, а второй — оператор присваивания значения переменной DO10I. Но разница между ними станет понятна транслятору только после того, как он дойдет до символа между цифрами 1 и 5, то есть до запятой в первом случае и точки — во втором, так как в языке FORTRAN пробелы игнорируются.

Синтаксический анализ, или разбор. Вторым этапом трансляции является *синтаксический анализ*, или *разбор*. На этом этапе лексемы, полученные в результате лексического анализа, используются для идентификации более крупных программных структур: операторов, объявлений, выражений и т. п. Синтаксический анализ обычно чередуется с семантическим. Сначала синтаксический анализатор идентифицирует последовательность лексем, формирующих синтаксическую единицу — выражение, оператор, вызов подпрограммы или объявление. Затем вызывается семантический анализатор для обработки этой синтаксической единицы. Обычно для связи синтаксического и семантического анализаторов используется стек. Синтаксический анализатор заносит в стек различные элементы обнаруженной синтаксической единицы, а затем они извлекаются из стека семантическим анализатором и обрабатываются. Основная задача, стоящая перед разработчиками языков в этой области, заключается в том, чтобы найти эффективные методы синтаксического анализа, в частности, использующие теорию формальных грамматик (см. раздел 3.3.1).

Семантический анализ. Семантический анализ является, возможно, самым важным этапом трансляции. На этом этапе обрабатываются структуры, которые были идентифицированы синтаксическим анализатором, и начинает формироваться структура выполняемого объектного кода. Таким образом, семантический анализ является мостом, соединяющим две части трансляции — анализ и синтез. На этом этапе осуществляется также ряд других важных вспомогательных функций, в том числе поддержка таблицы символов, обнаружение большинства ошибок, заме-

на макросов их определениями и выполнение операторов времени компиляции. В случае простых трансляций семантический анализатор может создать готовый к выполнению объектный код, но чаще результатом работы семантического анализатора становится некая внутренняя форма окончательной выполняемой программы, которая оптимизируется транслятором, прежде чем будет сгенерирован настоящий выполняемый код.

Семантический анализатор обычно разделен на ряд более мелких анализаторов, каждый из которых обрабатывает какую-либо определенную программную конструкцию. Эти анализаторы взаимодействуют между собой при помощи информации, хранящейся в различных структурах данных, в частности в центральной таблице символов. Например, семантический анализатор, обрабатывающий объявления простых переменных, занимается в основном тем, что просто заносит объявленные типы в таблицу символов. Затем семантический анализатор, обрабатывающий арифметические выражения, может использовать описанные типы для генерации соответствующих операций, присущих данному типу, в объектом коде. Конкретные функции семантических анализаторов значительно варьируются в зависимости от языка и логической организации транслятора. Некоторые наиболее общие функции можно описать следующим образом.

1. *Поддержка таблицы символов.* Таблица символов — это одна из центральных структур данных для каждого транслятора. Таблица символов состоит из элементов, каждый из которых соответствует какому-либо идентификатору, имеющемуся в программе, при этом в таблице отражены все такие идентификаторы. Лексический анализатор создает начальные элементы таблицы по мере того, как он сканирует исходную программу. Но элементы таблицы символов — это не просто идентификаторы; каждый элемент содержит также дополнительную информацию об атрибутах этого идентификатора: его тип (простая переменная, имя массива, имя подпрограммы, формальный параметр и т. д.), тип значения (целочисленный, вещественный и т. д.), та или иная среда ссылок — и вообще всю информацию, которую можно извлечь из объявления и применения этого идентификатора в исходной программе. Семантический анализатор заносит эту информацию в таблицу символов по мере обработки объявлений, заголовков подпрограмм и операторов программы. Другие части транслятора используют эту информацию для создания эффективного машинного кода.

   Таблица символов в трансляторах компилируемых языков обычно уничтожается после окончания трансляции. Но она может сохраняться и во время выполнения программы (например, в языках, которые допускают введение новых идентификаторов во время выполнения или отладки). Во всех реализациях языков ML, Prolog и LISP таблица символов создается во время трансляции и затем используется во время выполнения программы как центральная структура данных, определенная системой. В UNIX имеется весьма популярная программа dbx, которая использует таблицу символов для отладки программ на языке C.

2. *Включение неявной информации.* Часто бывает так, что информация, неявно содержащаяся в исходной программе, должна стать явной в объектной про-

грамме низкого уровня. Большая часть такой неявной информации относится к так называемым *соглашениям по умолчанию*, которые представляют собой определенные интерпретации, вступающие в силу в том случае, когда программист не задает явные спецификации. Например, в FORTRAN, если тип переменной, используемой в программе, не определен в объявлении, то по умолчанию этой переменной приписывается некоторый тип в зависимости от начального символа в ее имени.

3. *Обнаружение ошибок.* И синтаксический, и семантический анализаторы должны быть готовы к тому, что им придется взаимодействовать не только с корректными программами, но и с такими, в которых встречаются ошибки. В любой момент лексический анализатор может послать синтаксическому анализатору какую-нибудь лексему, которая не вписывается в окружающий контекст (например, разделитель операторов в середине выражения, объявление среди последовательности оператора, символ операции на том месте, где ожидалось появление идентификатора). Ошибка может быть менее заметной: например, вещественная переменная вместо требуемой целой переменной или индексированная переменная с тремя индексами вместо элемента двухмерного массива. Множество подобных ошибок может быть обнаружено на каждом этапе трансляции. Семантический анализатор должен не только распознавать встречающиеся подобные ошибки и выдавать соответствующие сообщения о них, но также должен во всех таких случаях (кроме самых серьезных) определять, каким образом можно продолжить синтаксический анализ оставшейся части программы.
4. *Макрообработка и операции, выполняемые во время компиляции.* Эти функции предусмотрены не во всех языках, но если они присутствуют, то обычно соответствующая обработка проводится семантическим анализатором.

Макрос в простейшей форме — это часть текста программы, которая определена отдельно и должна быть вставлена в программу во время трансляции, когда в программе встречается соответствующий вызов. Таким образом, макрос напоминает подпрограмму, с тем лишь отличием, что подпрограмма транслируется отдельно и вызывается во время выполнения (то есть связывание имени подпрограммы с ее семантикой происходит во время выполнения), а в случае с макросом вместо каждого его вызова во время трансляции просто подставляется его тело (то есть связывание происходит во время трансляции). Макросы могут являться простыми строками, подставляемыми в месте их вызова (например, в случае вызова макроса `PI` всегда подставляется значение `3.1416`). Но чаще макросы похожи на подпрограммы с параметрами, которые требуется обработать, прежде чем произойдет подстановка в месте вызова макроса.

В тех языках, в которых допускаются макросы, семантический анализатор должен идентифицировать вызовы макросов в исходной программе и осуществить соответствующую подстановку тела макроса. В таком случае часто приходится прерывать работу лексического и синтаксического анализаторов и переключать их на анализ строки, представляющей тело макроса, и лишь после этого продолжать анализ оставшейся части исходной программы. В другом варианте тело макроса может быть уже частично оттранслировано, так что семантический анализа-

тор может обрабатывать его непосредственно, вставляя соответствующий объектный код и создавая соответствующие элементы таблицы, а затем продолжить анализ исходной программы.

Операция, выполняемая во время компиляции, — это операция, которая должна выполняться во время трансляции и осуществлять контроль над трансляцией исходной программы. В языке С имеется несколько таких операций. Операция #define позволяет вычислять значение констант и выражений до начала компиляции программы. Конструкция #ifdef (if-defined) позволяет транслировать какой-то один из нескольких альтернативных фрагментов кода в зависимости от наличия или отсутствия определенных переменных. Эти переключатели позволяют программисту менять последовательность компилируемых операторов. Например, для компиляции альтернативных версий программ можно использовать содержащий текст исходной программы файл, подобный следующему:

```
#define pc                          /*выбор между версиями кода для PC и для UNIX*/
...
Program Write(...)
 #ifdef pc                          /*если определено pc, то требуется код для PC*/
 ...                                /*выполнение кода версии PC*/
                                    /*например, вывод в системе Microsoft Windows*/
 #else
 ...                                /*выполнение кода версии UNIX*/
                                    /*например, вывод в оболочке Motif X Windows*/
 #endif
```

3.2.2. Синтез объектной программы

На заключительном этапе трансляции происходит создание выполняемой программы на основе того, что было сделано семантическим анализатором. Этот этап обязательно включает генерацию кода и может также включать оптимизацию получившейся программы. Если подпрограммы транслировались отдельно или использовались библиотечные подпрограммы, то необходима заключительная стадия редактирования связей и загрузки, чтобы получить полностью пригодную для выполнения программу.

Оптимизация. Обычно результатом работы семантического анализатора является оттранслированная выполняемая программа, представленная в некотором *промежуточном коде*, внутреннее представление которого может быть цепочкой операций и операндов или таблицей, содержащей последовательности операций и операндов. На основе этого внутреннего представления генераторы кода могут создавать выходной объектный код в нужном формате. Тем не менее еще до генерации этого кода обычно происходит некоторая оптимизация программы в ее внутреннем представлении. Для семантического анализатора типичной ситуацией является генерация внутреннего представления программы по частям, как если бы анализировались отдельные фрагменты исходной программы. Обычно семантическому анализатору безразлична та часть кода, которая уже сгенерирована. Но такая фрагментарная организация работы анализатора может привести к созданию чрезвычайно неэффективного кода (например, в регистр, в котором может быть сохранено значение из некоторой области памяти в конце одного сгенерированного фрагмента, немедленно перезагружается значение из

той же самой области памяти в начале следующего фрагмента). Например, оператор

```
A = B + C + D
```

может быть сгенерирован в такой промежуточный код:

```
(a) Temp1 = B + C
(b) Temp2 = Temp1 + D
(c) A = Temp2
```

Его непосредственный перевод приведет к созданию следующего неэффективного кода:

```
1. Загрузить B в регистр (из (a))
2. Добавить C к этому регистру
3. Сохранить значение регистра в Temp1
4. Загрузить в регистр Temp1 (из (b))
5. Добавить D к этому регистру
6. Сохранить значение регистра в Temp2
7. Загрузить в регистр Temp2 (из (c))
8. Сохранить значение регистра в A
```

Инструкции 3, 4, 6 и 7 являются лишними, так как все промежуточные данные можно хранить в регистре, а потом сохранить результат в A. Обычно оказывается предпочтительным позволить семантическому анализатору строить такие неэффективные последовательности команд, чтобы потом на стадии оптимизации заменить их другими, лишенными таких очевидных недостатков.

Многие компиляторы не ограничиваются простой оптимизацией подобного сорта, а анализируют программу для выявления других возможностей усовершенствования (например, однократное вычисление общих подвыражений, вынесение инвариантных операций из тела цикла, оптимизация использования регистров, оптимизация вычисления формул доступа к элементам массива). Вопросам оптимизации посвящены многочисленные исследования, разработано много сложных и тонких методов оптимизации (см. библиографию в конце этой главы).

Генерация кода. После оптимизации оттранслированной во внутренний код программы она должна быть преобразована в инструкции ассемблера или в машинный код, или в какую-либо другую форму объектного кода, которая и станет результатом трансляции. Процесс генерации включает в себя определенное форматирование той информации, которая содержится во внутреннем представлении программы. Получившийся при генерации код либо может быть уже непосредственно выполняемым кодом, либо его придется подвергнуть дальнейшей обработке (например, может потребоваться обработка ассемблером или редактирование связей и загрузка).

Редактирование связей и загрузка. Этот заключительный этап трансляции не является обязательным. Фрагменты кода, полученные в результате раздельной трансляции подпрограмм, на данном этапе объединяются в единую выполняемую программу в ее окончательном виде. Выполняемые программы, полученные на предыдущих этапах трансляции, как правило, имеют почти окончательный вид. Исключение составляют те места, где имеются обращения к внешним данным или другим подпрограммам. Связи между этими разрозненными фрагментами задаются в присоединенных *таблицах загрузчика*, созданных транслятором. *За-*

грузчик с редактированием связей (или *редактор связей*) загружает различные фрагменты оттранслированного кода в память и затем использует присоединенные таблицы загрузчика для связывания этих фрагментов в единую программу, вставляя соответствующие данные и адреса подпрограмм в код по мере надобности. В итоге получается выполняемая программа в окончательной форме, готовая к работе.

Раскрутка

Часто транслятор для какого-либо нового языка пишется на этом же языке. Например, исходный компилятор для Pascal был написан на Pascal и предназначался для работы на виртуальной машине, использующей P-код. При этом возникает естественный вопрос: с чего начинать? Если исходный компилятор для Pascal был написан на Pascal, то как он сам был откомпилирован? Эта ситуация носит название *раскрутки*. Один из способов решения этой проблемы — откомпилировать компилятор вручную в его P-код. Это довольно скучное занятие, хотя и не слишком сложное. Когда компиляция закончена, интерпретатор P-кода может выполнить такой оттранслированный вручную компилятор. После того как появился работающий компилятор языка Pascal, можно произвести оптимизацию сгенерированного кода для улучшения самого процесса компиляции. Переход на новую машину не представляет проблемы, поскольку уже имеется исходный компилятор на предыдущей машине, который можно использовать для получения P-кода. Потребуется также изменить интерпретатор P-кода, но эта задача также не очень сложна.

Диагностика ошибок при помощи компилятора

В 60-е гг., когда пакетная обработка вызывала большие задержки в получении результатов компиляции, стала популярной диагностика ошибок при помощи компиляторов. Во врезке «Обзор языка 3.2» представлен краткий обзор работ в области развития компиляторов, диагностирующих ошибки, для CORC [31], CUPL и PL/C [32], проводившихся в Корнельском университете (Cornell University). В данном случае преследовалась цель создать быстро работающий компилятор, поскольку выполнение учебных программ обычно занимало доли секунды и после правильного выполнения о них можно было забыть.

3.3. Формальные модели трансляции

Как было отмечено в предыдущем разделе, часть теории компиляции, относящаяся к распознаванию синтаксических структур, достаточно стандартна и основывается, как правило, на теории контекстно-свободных языков. Мы посвятим несколько страниц краткому обзору этой теории. Формальное определение синтаксиса языка программирования называется обычно *грамматикой* по аналогии с общепринятой терминологией для естественных языков. Грамматика состоит из набора правил (называемых *правилами подстановки*, или *продукциями*[1]), определяю-

---

[1] В отечественной литературе этот термин употребляется довольно редко. — *Примеч. науч. ред.*

щих последовательности символов (или лексем), которые образуют допустимые для определяемого языка программы. *Формальная грамматика* — это грамматика, в которой используется строго определенная система обозначений. В технологии компиляции находят применение два класса грамматик: *НФБ-грамматика* (или контекстно-свободная грамматика) и *регулярная грамматика*, описанию которых и посвящены следующие ниже разделы. Краткий обзор других классов грамматик, которые, хотя и не столь полезны для развития теории компиляции, все же очень важны для понимания возможностей вычислительных машин, представлен в разделе 4.1.

Обзор языка 3.2. CORC, CUPL и PL/C

Характеристика. Компиляторы, способные автоматически исправлять ошибки в некорректных программах.

История. Во времена пакетной обработки (с начала 60-х и до середины 70-х гг.) зачастую требовался целый день, чтобы получить из вычислительного центра результаты компиляции программы. Следовательно, ошибки при компиляции обходились очень дорого. Чтобы приблизить момент начала вычислений по программе, в Корнельском университете (Cornell University) под руководством Ричарда Конвея (Richard Conway) была разработана серия компиляторов (CORC — Cornell Compiller, Корнельский компилятор; CUPL — Cornell University Programming Laguage, язык программирования Корнельского университета, и PL/C — Корнельская версия PL/I), которые автоматически диагностировали и исправляли некоторые простые синтаксические и семантические ошибки. Эта система была достаточно эффективна в отношении исправления простых ошибок. С развитием в 70-е гг. систем, работающих в режиме разделения времени, и появлением в 80-е персональных компьютеров и рабочих станций необходимость в автоматическом исправлении ошибок отпала.

Пример

```
/* PL/C Пример автоматического исправления ошибок в ПЛ/I */
SAMPLE:PROCEDURE OPTIONS(MAIN);        /*Главная процедура*/
          DECLARE VAR1 FIXED BINARY    /*VAR1 целая переменная; Добавляет отсутствующий
                                         символ ; в конце оператора*/
DECLARE VAR2 FIXED BINARY;
VAR1 = 1;
VAR2 = 2;
IF (VAR1 < VAR2                        /*Добавляет пропущенную скобку ) в конце строки*/
      THEN PUT SKIP LIST(VAR1, 'less than', VAR2)   /*Добавляет ; в конце оператора*/
                                       /*Печать: 1 less then 2*/
                                       /*по SKIP переходим на новую строку*/
      ESLE PUT SKIP LIST(VAR2, 'less than', VAR1);
DO WHILE(VAR2>0);
      PUT SKIP('Counting down', VAR2);/*Вставляет пропущенное ключевое слово LIST*/
      VAR2 VAR2 - 1;                   /*Вставляет знак равенства = после первого
                                         идентификатора VAR2*/
      END;                             /*Конец цикла*/
END;                                   /*Конец процедуры SAMPLE*/
```

Ссылки. R. W. Conway and T. Wilcox, "Design and Implementation of a Diagnostic Compiler for PL/I", *Comm. ACM* (16)3 (1973), 169–179.

3.3.1. НФБ-грамматика

Когда мы рассматриваем структуру английского предложения, обычно описываем его как последовательность категорий. То есть простое повествовательное предложение обычно представляется в виде:

подлежащее / глагол / дополнение.

Например:

The girl / ran / home (*Девочка побежала домой*).
The boy / cooks / dinner (*Мальчик готовит обед*).

Каждая из категорий может подразделяться на подкатегории. В приведенных примерах *подлежащее* состоит из *существительного* с *артиклем*. С учетом этого подразделения структура предложений такова:

артикль / существительное / глагол / дополнение.

Существуют другие структуры предложения кроме простейшей структуры представленных здесь повествовательных предложений. Например, простое вопросительное предложение в английском языке формируется следующим образом:

вспомогательный глагол / подлежащее / сказуемое,

что можно наблюдать в следующих предложениях

Did / the girl / run home (Побежала ли девочка домой)?
Is / the boy / cooking dinner (Готовит ли мальчик обед)?

Мы можем представить рассмотренные нами предложения набором правил. Можно сказать, что предложение может быть простым повествовательным или простым вопросительным, или выразить это с помощью специальной нотации:

<предложение> ::= <повествовательное> | <вопросительное>,

где символ ::= обозначает «определено как», а символ | обозначает «или». Дальнейшее определение предложений указанного типа может выглядеть следующим образом:

<повествовательное> ::= <подлежащее> <глагол> <дополнение>.
<подлежащее> ::= <артикль> <существительное>.
<вопросительное> ::= <вспомогательный глагол> <подлежащее> <сказуемое>?

Такая специфическая запись называется НФБ (нормальной формой Бэкуса, или формой Бэкуса–Наура). Она была разработана Джоном Бэкусом [14] в конце 50-х гг. как способ выражения синтаксического определения языка ALGOL. Питер Наур являлся председателем комитета, занимавшегося разработкой ALGOL. Приблизительно в то же самое время Ноам Хомский предложил подобную грамматическую форму, называемую *контекстно-свободной грамматикой* [27], для определения синтаксиса естественного языка. НФБ и формы контекстно-свободной грамматики эквивалентны по своим возможностям, различия касаются только системы обозначений. По этой причине при обсуждении вопросов синтаксиса термины *НФБ-грамматика* и *контекстно-свободная грамматика* обычно являются взаимозаменяемыми.

Синтаксис

НФБ-грамматика состоит из конечного набора правил НФБ-грамматики, которые определяют *язык* — в нашем случае язык программирования. Прежде чем подробно рассматривать эти правила, следует разобраться с термином «*язык*». Поскольку синтаксис имеет отношение к форме (структуре), а не к значению (смыслу), то язык (программирования) с точки зрения синтаксиса представляет собой множество *синтаксически правильных программ*, каждая из которых есть просто последовательность символов. В отношении семантики синтаксически правильная программа может не иметь никакого смысла (то есть при выполнении она не обязательно должна производить какие-нибудь полезные вычисления, даже более того, она может вообще ничего не вычислять).

Например, если вернуться к определенным выше повествовательным и вопросительным предложениям, то в приводимом ниже предложении синтаксис *подлежащее / глагол / дополнение* полностью соблюден:

The home / ran / girl (*Дом побежал девочка*[1]),

но получившееся в результате предложение не имеет смысла[2].

Что касается синтаксиса языков программирования, мы можем еще больше абстрагироваться от смысла синтаксических последовательностей и определить язык как *множество цепочек символов конечной длины*, причем символы выбираются из определенного конечного алфавита. При таком определении языком можно назвать:

1) множество всех операторов присваивания в C;
2) множество всех программ на C;
3) множество всех атомов LISP;
4) множество последовательностей из элементов a и b, таких, что все элементы a в каждой последовательности предшествуют элементам b (например, a, aab, abb, ...).

Язык может состоять как из конечного множества цепочек (например, язык, составленный из всех разделителей языка Pascal: begin, end, if, then и т. д.), так и из бесконечного множества цепочек (например, из цепочек, описанных в п. 4). Единственным ограничением является то, что длина цепочек должна быть конечной, а символы должны быть взяты из определенного конечного множества (алфавита).

Если внимательно рассмотреть приведенные выше примеры определения языков, то можно обнаружить некоторые сложности, возникающие при описании языка

[1] В английском языке последовательность членов предложения достаточно строго определена, то есть роль слова в предложе[illegible]т от местоположения этого слова. Так, дополнение не может стоять перед подлежащим, поэтому в приведенном примере слово home однозначно должно играть роль подлежащего. — *Примеч. пер.*

[2] В связи с этими рассуждениями можно привести в качестве примера научный эксперимент ученого-лингвиста академика Л. В. Щербы, который просил своих слушателей найти подлежащее и сказуемое в известной фразе: «Глокая куздра штеко будланула бокра и кудрячит бокренка» (см.: *Яглом И. М.* Математические структуры и математическое моделирование. М.: Сов. радио, 1980. 144 с.). — *Примеч. науч. ред.*

программирования с помощью естественного языка (русского или английского в данном случае). Рассмотрим снова четвертый пример. Принадлежит ли цепочка, составленная из единственного символа b, этому языку? Мы знаем, что в цепочке все символы a (которых в данном случае у нас просто нет) должны предшествовать символам b, но должна ли цепочка обязательно содержать хотя бы один символ a? Аналогично, принадлежит ли цепочка, составленная из единственного символа a, определенному нами языку? Таким образом, мы видим, что имеющееся описание языка является неполным.

Эта проблема может быть решена при помощи множества формальных математических правил, точно определяющих, какие цепочки допустимы в языке. В простейшем случае грамматическое правило может быть задано простым перечислением элементов конечного языка, например:

```
<цифра> ::= 0|1|2|3|4|5|6|7|8|9
```

Это правило простым перечислением набора возможностей определяет язык, состоящий из десяти цепочек, содержащих единственный символ: 0, 1, 2, 3, 4, 5, 6, 7, 8 или 9. Это грамматическое правило читается так: «*Цифра* – это либо 0, либо 1, либо 2, либо...» Термин *цифра* называется *синтаксической категорией*, или *нетерминальным символом*[1], и служит именем для языка, определяемого данным синтаксическим правилом. Символы, из которых образуются цепочки в языке, – в нашем случае цифры от 0 до 9, называются *терминальными символами*. Часто символ ::= заменяется на →, особенно в тех случаях, когда нетерминальные символы записываются в виде одиночных букв в верхнем регистре (например, правило <X> ::= <B>|<C> часто записывается как X → B|C). В этой книге используются оба обозначения.

Определив основной набор терминальных символов, можно использовать их для конструирования более сложных цепочек. Например, правило

```
<условный оператор> ::=
if <булево выражение> then <оператор> else <оператор>
        | if <булево выражение> then <оператор>
```

определяет язык, состоящий из конструкций <условный оператор>, определенных при помощи терминальных символов <булево выражение> и <оператор>, которые, в свою очередь, определяются другими правилами грамматики. Обратите внимание на то, что приведенное правило порождает два возможных вида условных операторов (разделенных вертикальной чертой |). Каждый вариант получается посредством конкатенации различных элементов, которыми могут быть литеральные цепочки (например, if или else) или синтаксические категории. Когда в правиле указывается некоторая синтаксическая категория, это означает, что в этом месте может быть использована любая цепочка символов языка, определяемого данной категорией. Например, если предположить, что категория <булево выражение> состоит из набора строк, представляющих собой некоторые логические выражения, то любое из этих выражений может быть вставлено между if и then в условном операторе.

[1] В отечественной литературе чаще используется понятие «нетерминальный символ», или просто «нетерминал». — *Примеч. науч. ред.*

Другой тип грамматических правил использует рекурсивное определение синтаксической категории. Например, рекурсивное правило

<целое без знака> ::= <цифра>|<целое без знака><цифра>

определяет <целое без знака> как последовательность элементов <цифра>. В первом из двух вариантов <целое без знака> определяется как одна цифра, а во втором варианте к этой исходной цифре добавляется вторая, третья и т. д.

Полная НФБ-грамматика — это просто набор подобных правил, которые в совокупности определяют иерархию языков, завершающуюся синтаксической категорией самого верхнего уровня, которая в случае языков программирования называется <программой>. В табл. 3.1 представлена более сложная грамматика, определяющая синтаксис класса простых операторов присваивания, в которой в качестве основных синтаксических категорий (предполагается, что они определены ранее) используются <идентификатор> и <число>.

Таблица 3.1. Грамматика для простых операторов присваивания

<оператор присваивания>::=	<переменная> = <арифметическое выражение>
<арифметическое выражение>::=	<терм> | <арифметическое выражение> + <терм> | <арифметическое выражение> –<терм>
<терм>::=	<первичное выражение> | <терм> x <первичное выражение> | <терм> / <первичное выражение>
<первичное выражение>::=	<переменная> | <число> | (<арифметическое выражение>)
<переменная>::=	<идентификатор> | <идентификатор> [<список индексов>]
<список индексов>	<арифметическое выражение> | <список индексов>, <арифметическое выражение>

Деревья грамматического разбора

Имея некоторую грамматику, мы можем последовательно использовать правила подстановки для генерации цепочек нашего языка. Например, следующая грамматика генерирует все последовательности (цепочки), состоящие из сбалансированных круглых скобок (то есть такие последовательности, в которых каждой открывающей скобке соответствует закрывающая):

$$S \to SS|(S)|()$$

Любую цепочку мы можем преобразовать путем замены любого нетерминального символа на правую часть любого правила подстановки, в которой этот нетерминальный символ имеется в левой части. Например, цепочку (()()) мы можем получить из S следующим образом:

1) заменяем S по правилу S → (S) и получаем (S);
2) заменяем S в (S) по правилу S → SS и получаем (SS);
3) заменяем первое S в (SS) по правилу S → () и получаем (()S);
4) заменяем S в (()S) по правилу S → () и получаем (()()).

Если использовать символ ⇒ для указания того, что одна цепочка выводима из другой, весь вывод можно записать следующим образом:

$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (()S) \Rightarrow (()())$$

Каждый член этого вывода является *сентенциальной формой*, и мы формально определяем язык как множество сентенциальных форм, которые состоят только из терминальных символов[1] и выводимы из исходного символа грамматики.

Использование формальной грамматики для определения синтаксиса языка программирования важно как для программистов, использующих этот язык, так и для его разработчиков. Пользователь может получить из нее ответы на сложные вопросы относительно вида программы, пунктуации и структуры. Разработчик может использовать грамматику для того, чтобы определить все допустимые в данном языке структуры исходных программ, с которыми, возможно, придется взаимодействовать транслятору. И программист, и разработчик имеют общее, согласованное определение, которое можно использовать для разрешения споров о допустимых синтаксических конструкциях. Формальное определение синтаксиса помогает также устранению незначительных синтаксических различий между отдельными реализациями языка.

Чтобы определить, представляет ли данная цепочка синтаксически правильную программу на языке, определяемом НФБ-грамматикой, нужно использовать грамматические правила для проведения синтаксического анализа или грамматического разбора этой цепочки. Если разбор прошел успешно, то данная цепочка принадлежит указанному языку. Если же не удается провести ее грамматический разбор с помощью заданных правил грамматики, то она не принадлежит этому языку. На рис. 3.3 представлено дерево грамматического разбора, которое получилось в результате синтаксического анализа оператора присваивания W = Y × (U + V) с использованием НФБ-грамматики, правила которой представлены в табл. 3.1.

НФБ-грамматика сопоставляет каждой цепочке определяемого с ее помощью языка некоторую структуру, как показано на рис. 3.3. Заметим, что такой структурой всегда будет дерево ввиду ограничений, налагаемых на правила НФБ-грамматики. Листьями такого дерева являются отдельные символы или лексемы (лексические единицы) входной цепочки. Каждая промежуточная точка ветвления в дереве сопоставлена с некоторой синтаксической категорией, которая указывает класс, к которому принадлежит расположенное ниже ее поддерево. Корню дерева соответствует синтаксическая категория, указывающая на весь язык, — в нашем случае это категория <оператор присваивания>.

Дерево грамматического разбора предоставляет интуитивную семантическую структуру для большей части программы. Так, например, НФБ-грамматика для языка Pascal определяет структуру любой программы в виде последовательности объявлений и операторов со вложенными блоками. Структура операторов, в свою очередь, состоит из разного рода выражений, а сами эти выражения составлены из простых и индексированных переменных, примитивных операций, вызовов функций и т. д. На самом нижнем уровне даже идентификаторы и числа разлагаются на свои составные части. При изучении грамматики программист получает возможность глубже понять различные структуры, из которых составляются синтаксически правильные программы. Важно отметить, что не всегда в грамматике данному элементу програм-

---

[1] Вывод завершается, если в цепочке больше нет ни одного нетерминала. Отсюда и название «терминальный символ» (от англ. terminal — заключительный, конечный), в том смысле, что на нем завершается вывод. — *Примеч. науч. ред.*

мы присваивается та структура, которая кажется для него естественной. Один и тот же язык может быть определен множеством различных грамматик, как можно легко заметить, проведя некоторые манипуляции с той грамматикой, что приведена в табл. 3.1. Например, в табл. 3.2 задана грамматика, определяющая тот же язык, что и грамматика в табл. 3.1, но заметьте, что структуры, определяемые этой новой грамматикой, далеки от структур, которые можно было бы определить интуитивно.

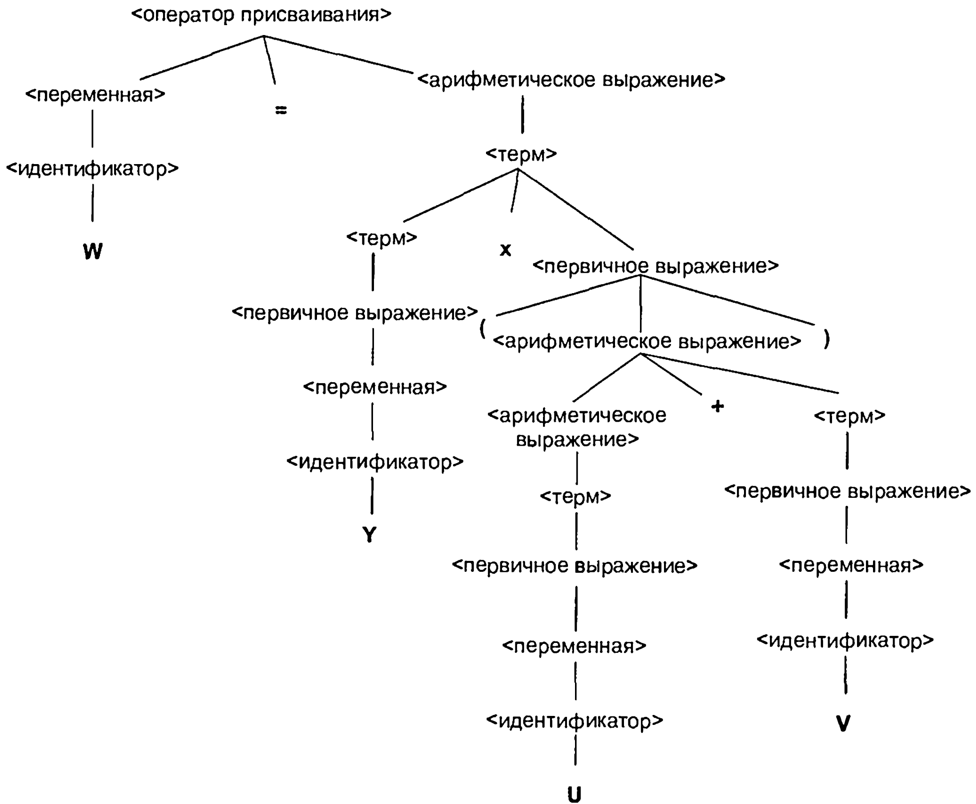


Рис. 3.3. Дерево грамматического разбора для оператора присваивания

Таблица 3.2. Альтернативная НФБ-грамматика

<оператор присваивания>::=	<переменная> = < арифметическое выражение >
<арифметическое выражение>::=	<терм> | <арифметическое выражение> x <терм> | <арифметическое выражение> + <терм>
<терм>::=	<первичное выражение> | <терм> – <первичное выражение> | <терм> / <первичное выражение>
<первичное выражение>::=	<переменная> | <число> | (<арифметическое выражение>)
<переменная>::=	<идентификатор> | <идентификатор> [<список индексов>]
<список индексов>	<арифметическое выражение> | <список индексов>, <арифметическое выражение>

Несмотря на чрезвычайно простую структуру НФБ-грамматики, она может быть с успехом использована для определения синтаксиса большинства языков программирования. Только та область синтаксиса, которая связана с контекстной зависимостью, не может быть определена при помощи этой грамматики. Не могут быть заданы с помощью одной лишь НФБ-грамматики такие ограничения, как, например:

- один и тот же идентификатор не может быть описан дважды в одном блоке;
- каждый идентификатор должен быть описан в каком-либо блоке, определяющем область его использования;
- на массив, определенный как двухмерный, нельзя ссылаться с помощью трех индексов.

Ограничения такого рода должны быть определены как дополнение к формальной НФБ-грамматике. В главе 4 мы обсудим некоторые формальные методы, которые помогают преодолеть недостатки модели НФБ.

Процесс использования НФБ-грамматики при построении дерева грамматического разбора для определенной программы хорошо изучен. В разделе 3.4 мы кратко опишем одну простую методику грамматического разбора — рекурсивный спуск — это поможет читателю составить представление о возникающих в связи с ним проблемах.

Неоднозначность

Как говорилось ранее, неоднозначность — это проблема синтаксиса. Рассмотрим фразу «They are flying planes». Мы можем представить ее двумя способами:

They / are / flying planes. (Они / являются / летящими самолетами.)
They / are flying / planes. (Они / летят / самолетами.)

Обе эти фразы имеют вполне определенное, хотя и различное, значение. В первом случае *They* (они) относится к самолетам, а слово *flying* — это причастие, служащее определением самолетов. Во втором случае *They* относится к кому-то, кто летит на самолете.

Неоднозначность часто является свойством не языка, а грамматики. Например, грамматика G_1, которая определяет всевозможные цепочки из нулей и единиц, является неоднозначной:

$G_1 : S \rightarrow SS \mid 0 \mid 1$

Мы знаем, что эта грамматика является неоднозначной, потому что в языке существуют цепочка, имеющая два различных дерева грамматического разбора (рис. 3.4).

Если любая грамматика для какого-либо языка является неоднозначной, то говорят, что язык обладает *наследственной неоднозначностью*. Но в нашем случае язык, состоящий из всех двоичных цепочек, не является наследственно неоднозначным, поскольку существует лишенная неоднозначности грамматика G_2, которая определяет те же цепочки:

$G_2 : T \rightarrow 0T \mid 1T \mid 0 \mid 1$

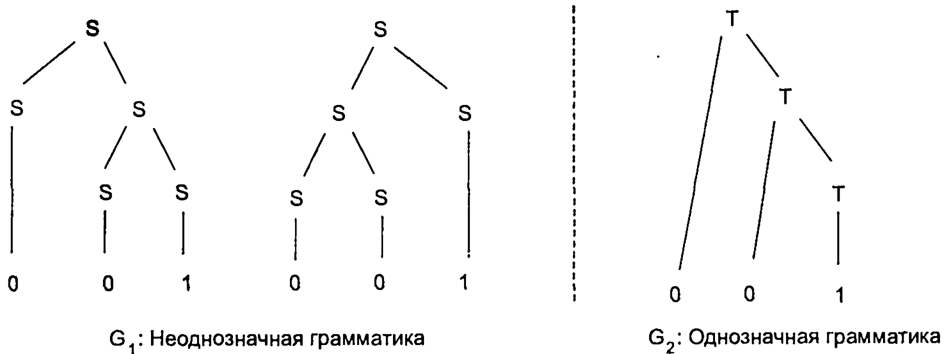


Рис. 3.4. Неоднозначность в грамматиках

Расширения НФБ-нотации

Несмотря на простоту, элегантность и мощность НФБ-грамматик, они не являются идеальным средством для сообщения программисту-практику правил конкретного языка программирования. Основная причина этого заключается в простоте правил НФБ-грамматики, которая приводит к весьма неестественному представлению общих синтаксических конструкций для необязательных, альтернативных и повторяющихся элементов какого-либо синтаксического правила. Например, чтобы выразить простую синтаксическую идею «*целое со знаком* есть последовательность цифр, начинающаяся с необязательного символа плюс или минус», в НФБ-грамматике придется написать довольно сложный ряд рекурсивных правил, а именно:

```
<целое со знаком> ::= +<целое> | -<целое>
<целое> ::= <цифра> | <целое><цифра>
```

Мы опишем некоторые расширения НФБ-нотации, которые позволяют избежать подобных неестественных способов определения простых синтаксических свойств некоторых грамматик.

Расширенная НФБ-нотация. Для расширения НФБ-нотации применяются следующие дополнительные обозначения, которые не ограничивают возможности НФБ-грамматики, но упрощают описания языков:

- необязательный элемент может быть обозначен заключением его в квадратные скобки — [...];
- альтернативные варианты вводятся при помощи вертикальной черты | и в случае необходимости могут быть заключены в квадратные скобки ([,]);
- произвольная последовательность экземпляров одного и того же элемента может быть обозначена заключением его в фигурные скобки, за которыми следует символ «звездочка» — {...}*.

В качестве примеров рассмотрим следующие правила:

```
целое со знаком: <целое со знаком> ::= [+|-]<целое>{<целое>}*
идентификатор: <идентификатор> ::= <буква>{<буква>|<цифра>}*
```

В качестве другого примера можно привести более интуитивное определение арифметических выражений, как они описаны в табл. 3.1, с использованием расширенной НФБ-нотации.

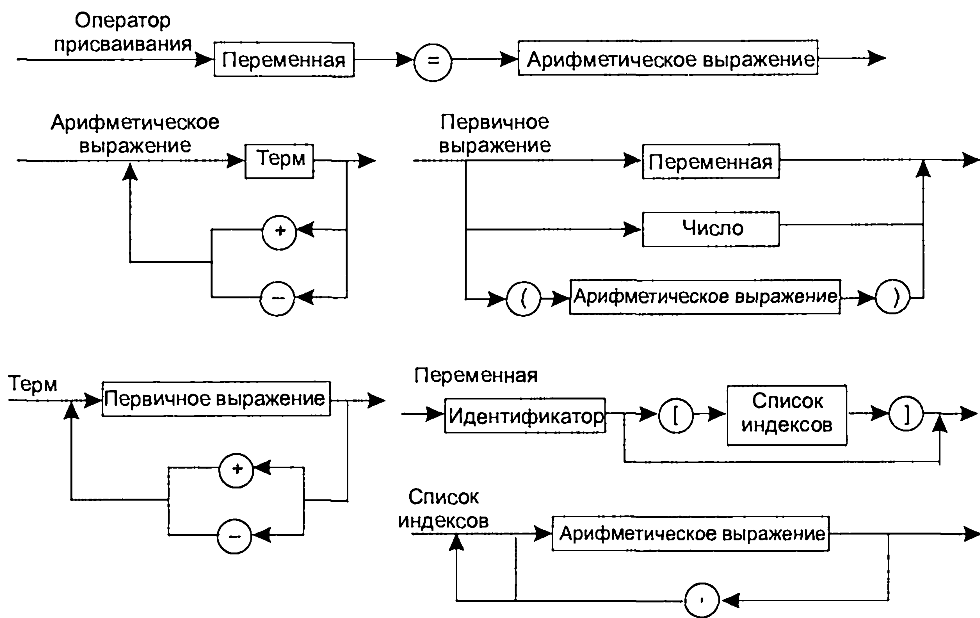


Рис. 3.5. Синтаксические схемы для простых операторов присваивания

Такие правила, как

<арифметическое выражение> ::= <терм> | <арифметическое выражение> + <терм>

отражают тот факт, что арифметическое выражение может состоять из произвольного количества термов. Используя расширенную НФБ-нотацию, можно это выражение записать так:

<арифметическое выражение> ::= <терм> { + <терм>}*

Вся грамматика арифметических выражений может быть переформулирована при помощи расширений НФБ-нотации. Результат представлен в табл. 3.3.

Таблица 3.3. Расширенная НФБ-грамматика для простых операторов присваивания

<оператор присваивания> ::=	<переменная> = <арифметическое выражение>
<арифметическое выражение> ::=	<терм> {[+ | –] <терм>}*
<терм> ::=	<первичное выражение> {[× | /]<первичное выражение>}*
<первичное выражение>: :=	<переменная> | <число> | (<арифметическое выражение>)
<переменная> ::=	<идентификатор> | <идентификатор> [<список индексов>]
<список индексов> :=	<арифметическое выражение> {,<арифметическое выражение>}*

Синтаксические схемы. Синтаксическая схема (называемая также *железнодорожной диаграммой*, поскольку она напоминает расположение стрелок на же-

лезной дороге) — это графический способ выражения правил грамматики с помощью расширенной НФБ-нотации. Каждое правило представляется в виде некоторой траектории от расположенной слева точки входа до расположенной справа точки выхода. Любая траектория от входа к выходу представляет цепочку, генерируемую этим правилом. Если другие правила мы представим в виде прямоугольников, а терминальные символы изобразим кружками, то для грамматики, представленной в табл. 3.3, получим синтаксические схемы, изображенные на рис. 3.5.

Например, чтобы получить синтаксическую категорию <терм>, следует двигаться слева от точки входа либо по траектории, проходящей через <первичное выражение> и выходящей справа через точку выхода, либо по траектории, также проходящей через <первичное выражение> и затем совершающей один или более циклов, проходя через кружки с операциями умножения или деления, а затем снова через <первичное выражение> и, наконец, также выходящей справа через точку выхода. Этому соответствует следующее правило, записанное с использованием расширенной НФБ-нотации:

```
<терм> ::= <первичное выражение> {[x | /]<первичное выражение>}*
```

3.3.2. Конечные автоматы

Все лексемы языка программирования имеют простую структуру. Как было сказано ранее, этап лексического анализа при компиляции заключается в том, что текст исходной программы разбивается на последовательность лексем. Теперь мы рассмотрим этот процесс с иной точки зрения, нежели в предыдущем разделе, посвященном НФБ-грамматике, а именно опишем машинную модель распознавания лексем.

Любой идентификатор должен начинаться с буквы; пока символы, следующие за ней, являются цифрами или буквами, они входят в имя этого идентификатора. Целое число — это просто последовательность цифр. Зарезервированное слово `if` — это последовательность из двух букв, `i` и `f`. Во всех подобных случаях для распознавания лексем применяется простая модель под названием *конечный автомат* (КА), или *машина состояний*. Пока мы знаем, в каком из состояний находимся, мы можем определить, является ли очередной вводимый символ частью лексемы, которую мы ищем.

На рис. 3.6 схематически изображен простой КА, задача которого — распознавать двоичные цепочки с нечетным количеством единиц. Процесс начинается с состояния A (содержащего дугу ввода, не выходящую из другого состояния), затем в зависимости от следующего введенного символа автомат переходит в одно из состояний, A или B, двигаясь вдоль дуги, помеченной этим символом. Очевидно, что состояние B (*конечное состояние*, обозначенное двойным кружком) соответствует нечетному количеству единиц во введенной цепочке. Поэтому, пока автомат находится в состоянии B, введенная на данный момент цепочка удовлетворяет предъявленному требованию и допускается этим автоматом. Когда автомат находится в состоянии A, такая цепочка не допускается.

Последовательность действий КА для ввода цепочки 100101 будет следующей:

Ввод	Текущее состояние	Допускается ли цепочка
null	A	Нет
1	B	Да
10	B	Да
100	B	Да
1001	A	Нет
10010	A	Нет
100101	B	Да

Мы видим, что после ввода 1, 10, 100 и 1001001 автомат попадает в состояние B, в цепочке содержится нечетное количество единиц и, таким образом, ввод допустим.

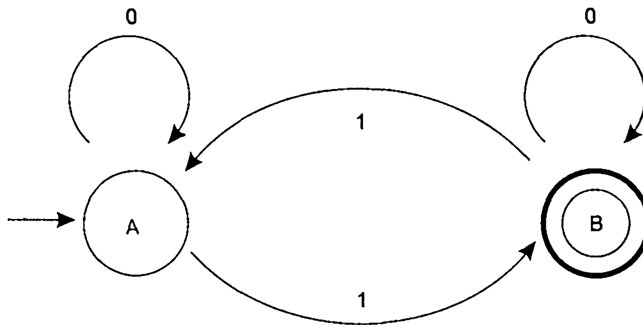


Рис. 3.6. КА, распознающий битовые цепочки с нечетным количеством единиц

Вообще для любого КА определено одно начальное состояние, как минимум одно конечное состояние и ряд переходов между ними (изображенных помеченными дугами). Говорят, что любая введенная цепочка символов, которая переводит автомат из начального состояния в конечное через ряд переходов, допускается этим автоматом.

На рис. 3.7 изображен другой КА, распознающий целые числа со знаком. Если использовать расширенную НФБ-нотацию, рассмотренную нами в предыдущем разделе, то с ее помощью можно определить такие числа, как <целое со знаком>::=[+ | -]<целое> {<целое>}*. Этот пример иллюстрирует еще одно важное свойство КА: между абстрактными машинами и грамматиками существует взаимное соответствие. В этой главе будет показано, что КА может быть смоделирован при помощи грамматики.

Недетерминированный конечный автомат. До сих пор при обсуждении автоматов подразумевалось, что все переходы между состояниями однозначно определены текущим состоянием и очередным введенным символом. То есть для каждого состояния КА и каждого введенного символа существует единственно возможный переход в некоторое следующее состояние (оно может совпадать или не совпадать с предыдущим). Подобный конечный автомат называется *детерминированным*. Если автомат имеет n состояний и входной алфавит состоит из k символов, то количество переходов (помеченных дуг) будет равно $n \times k$. На рис. 3.7 изображен такой

детерминированный конечный автомат. Отсутствие некоторых дуг (переходов) не представляет проблемы, так как мы всегда можем добавить дуги (переходы) к дополнительному поглощающему неконечному состоянию ошибки. Проблема недетерминированности состоит в том, что существует несколько одинаково помеченных дуг, исходящих из одного состояния. Таким образом, в недетерминированном автомате появляется возможность выбора между различными переходами.

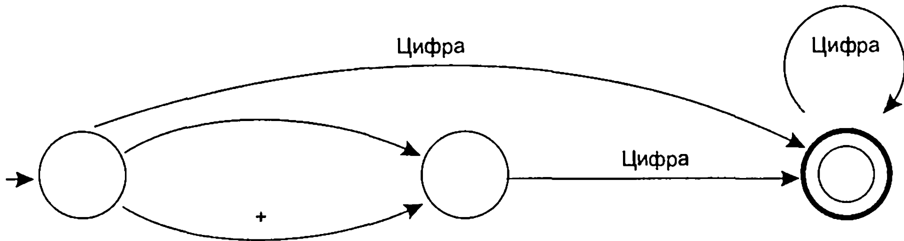


Рис. 3.7. КА, распознающий целые числа с необязательным знаком

Итак, *недетерминированный* конечный автомат определяется как конечный автомат, обладающий:

1) множеством состояний (узлы графа);
2) начальным состоянием (один из узлов);
3) множеством конечных состояний (подмножество узлов);
4) входным алфавитом (символы для маркировки дуг, определяющих переходы между состояниями);
5) множеством маркированных символами входного алфавита дуг, соединяющих узлы и определяющих переходы между состояниями.

В таком недетерминированном автомате из любого узла может выходить несколько дуг (или не выходить вовсе), причем одну и ту же маркировку могут иметь несколько дуг. Такие переходы недетерминированы, поскольку для одного введенного символа может быть несколько разных вариантов дальнейшего пути (см. пример на рис. 3.9). В таком случае считается, что некоторая цепочка *допускается* конечным автоматом, если существует *хотя бы один* путь от начального к какому-либо из конечных узлов, даже если другие пути, соответствующие той же цепочке, *не* приводят к конечному состоянию. В детерминированном случае автомат всегда будет находиться в определенном состоянии, которое однозначно зависит от вводимых символов. Дополнительные сведения о конечных автоматах вы можете найти в задаче 12 в конце этой главы.

Регулярные грамматики

Частным случаем НФБ-грамматик являются *регулярные грамматики*. Можно показать, что язык, определяемый некоторой регулярной грамматикой, эквивалентен языку, определяемому некоторым конечным автоматом. Правила регулярных грамматик записываются в следующей форме:

```
<нетерминальный символ> ::= <терминальный символ> <нетерминальный символ> |
    <терминальный символ>
```

Каждое правило состоит из терминального символа, за которым может следовать нетерминальный. Грамматика, генерирующая битовые цепочки, которые оканчиваются нулем, задается следующим образом:

```
A → 0A | 1A | 0
```

Первых два варианта используются для генерации произвольных битовых цепочек, а третий — для того, чтобы создать битовую цепочку, оканчивающуюся нулем.

Между КА и регулярными грамматиками существует тесная связь; можно показать, что они генерируют одни и те же языки (см. задачи и упражнения в конце этой главы).

Регулярные выражения

Регулярные выражения представляют собой третий способ описания языка, эквивалентный описанию с помощью регулярных грамматик и конечных автоматов. Регулярное выражение определяется рекурсивно следующим образом:

1) отдельные терминальные символы являются регулярными выражениями;
2) если a и b — регулярные выражения, то a∨b, ab, (a) и a* также являются регулярными выражениями;
3) ничто другое не является регулярным выражением.

ab — это результат конкатенации, или объединения, в последовательность регулярных выражений a и b, a∨b обозначает выбор одного из выражений, a или b, и наконец, a* называется замыканием Клини регулярного выражения a и представляет собой ноль или более повторений регулярного выражения a (например, сюда относится пустая строка, часто обозначаемая как ε, a, aa, aaa,...)[1].

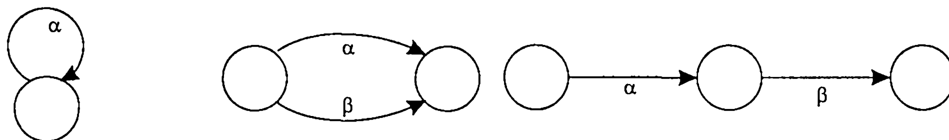


Рис. 3.8. Преобразование регулярных выражений в КА

Регулярные выражения можно использовать для представления любого языка, определенного регулярной грамматикой или конечным автоматом, хотя преобразование конечного автомата в регулярное выражение не всегда очевидно. Следующая небольшая таблица иллюстрирует использование регулярных выражений.

Язык	Регулярное выражение
Идентификаторы	буква(буква∨цифра)*
Битовые цепочки, кратные 2	(0∨1)*0
Битовые цепочки, содержащие 01	(0∨1)*01(0∨1)*

[1] Операция заключения регулярного выражения в круглые скобки, о которой авторы ничего не сказали, означает создание группы, к которой можно применить другие операции, например замыкание Клини. — *Примеч. науч. ред.*

Преобразование любого регулярного выражения в КА достаточно просто. Операция ∨ представляет альтернативные варианты перехода от состояния А к состоянию В, конкатенация означает последовательность состояний, а замыкание Клини соответствует циклам (рис. 3.8). На рис. 3.9 представлен конечный автомат для третьего регулярного выражения из таблицы. Заметим, что в результате мы получаем недетерминированный КА, который, однако, может быть преобразован в эквивалентный детерминированный КА (см. задачу 12). Как будет показано в разделе 3.3.3, несмотря на их сложность, регулярные выражения и конечные автоматы достаточно легки для понимания, а для упрощения обработки регулярных выражений был разработан такой инструмент, как язык программирования Perl.

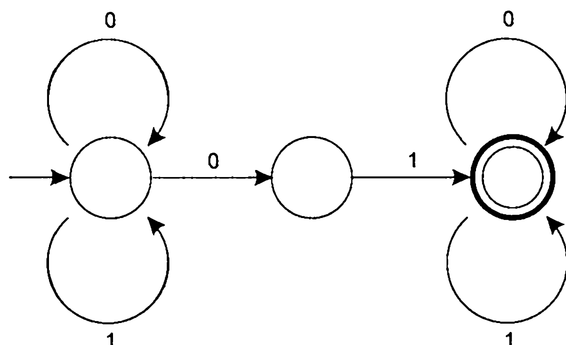


Рис. 3.9. Преобразование регулярного выражения (0∨1)*01(0∨1)* в КА

Вычислительные возможности КА

Конечный автомат способен содержать конечное количество информации — ограниченное множество состояний. Следовательно, количество распознаваемых им цепочек ограничено. Например, КА не способен распознать множество цепочек типа $a^n b^n$. Для доказательства допустим, что $a^n b^n$ распознается некоторым КА с числом состояний k. Тогда для любого $n > k$ при считывании символа a автомат должен будет войти в то же самое состояние по крайней мере дважды. Следовательно, какое-то подмножество этой цепочки вызывает цикл внутри КА. Это означает, что $a^n = wxy$, где подцепочка x вызывает циклический возврат КА в то же самое состояние. Легко видеть, что цепочка wx^*yb^n будет принята конечным автоматом. Принятыми цепочками будут цепочки вида $a^{m+i \times p}b^n$, где m и p — целые числа, а i — произвольное число; но это отнюдь не то же самое, что цепочка $a^n b^n$.

3.3.3. Обзор языка Perl

Perl — это язык программирования, хорошо приспособленный для работы с регулярными выражениями. Как сказано в разделе 1.4.2, операционная система UNIX ввела в практику системного программирования сценарий на языке командного интерпретатора shell, который можно рассматривать как язык программируемых процессов для управления выполнением программ на компьютере. Данными, обрабатываемыми этими сценариями, были программы и файлы, входящие в систему компьютера. В качестве вспомогательных средств для программирования на

языке командного интерпретатора shell были разработаны разнообразные языки для обработки строковых данных. Одними из первых были AWK, появившийся в 1977 г. (назван в честь своих авторов — Альфреда В. Ахо (Alfred V. Aho), Питера Дж. Вейнбергера (Peter. J. Weinberger) и Брайана В. Кернигана (Brian W. Kernighan)), и SED, потоковый редактор (stream editor), созданный по образцу одного из первых редакторов UNIX *ed*.

Поскольку AWK был предназначен для работы с файлами определенной регулярной структуры, его можно было использовать, например, следующим образом: взять файл, состоящий из имен и соответствующих им адресов электронной почты, и, используя AWK, разослать по всем этим адресам письма. Для этого нужно было написать shell-сценарий, содержащий цикл, на каждом шаге которого вызывался бы AWK для извлечения очередного адреса электронной почты из файла, после чего сообщение могло быть отправлено соответствующему адресату. Зная язык командного интерпретатора shell и возможности сопоставления с образцами AWK, программист достаточно легко мог создать подобный процесс обработки списка рассылки.

История. Для облегчения shell-программирования в UNIX было разработано много новых языков управления процессами типа AWK. Язык Perl (Practical Extraction and Report Language) был создан в 1986 г. Ларри Уоллом (Larry Wall) для решения задач управления конфигурацией сети, состоящей из нескольких компьютеров. Вначале Уолл использовал механизм под названием *B-news*, но он неадекватно выполнял задачу создания и обработки необходимых отчетов. AWK не мог одновременно открывать и закрывать множество файлов. В результате появился новый язык Perl, который унаследовал некоторые черты предшественников — языков AWK и SED, но более подходил для данного применения.

Сначала язык назывался PEARL (жемчуг), но поскольку так назывался существующий графический язык, было решено сократить название. В языке были предусмотрены скалярные данные, возможность сопоставления с образцом, управление и обработка файлов. Со временем появились новые версии этого языка, последняя включает возможность объектно-ориентированного программирования. Широкое распространение WWW привело к открытию, что Perl является одним из наиболее подходящих языков для программирования задач интерактивного взаимодействия в Web — задач обработки на сервере информации, введенной пользователем на web-странице.

Краткий обзор языка. Perl — это интерпретируемый язык, предназначенный для эффективной обработки текстов. Синтаксис языка Perl построен по образцу языка C, так как исходно Perl развивался как командный язык в операционной системе UNIX, где C являлся основным языком программирования. Ввиду сходства с C Perl столь же удобен (или неудобен — в зависимости от того, считаете ли вы C удобным для чтения) для чтения, как и C.

Переменные в языке Perl начинаются с символа $ и могут содержать как целые числа, так и строки. Также в Perl имеются массивы скаляров и ассоциативные массивы. Ассоциативные массивы называются также *массивами с адресацией по содержанию*, поскольку для доступа к информации можно пользоваться не только индексами массива, но и содержанием элемента. Например, в большин-

стве современных операционных систем с программой ассоциирована некоторая среда или окружение, содержащее переменные с определенными значениями. В языке Perl это окружение доступно при помощи специального ассоциативного массива $ENV. Одним из элементов этого окружения (в системе UNIX) является идентификатор (ID) пользователя. В Perl достаточно написать следующий оператор:

```
print "Пользователь этой программы: $ENV{'USER'}\n";
```

и на экране монитора будет напечатано имя пользователя (переменная окружения USER), причем не потребуется определять, где в окружении хранится информация об ID пользователя. Этот пример показывает, как легко Perl интегрируется в операционную систему для создания сценариев обработки процессов.

Простая программа Perl состоит из последовательности операторов print. Она также включает обычные последовательности структур управления наподобие циклов for, while и until и условного оператора if. Особенно удобен оператор foreach, который позволяет совершать цикл по всем элементам массива и выполнять для каждого из них какие-то действия, не зная заранее о размерах массива.

В Perl, как и в некоторые другие языки создания процессов, встроена возможность обрабатывать регулярные выражения. Логическая операция =~ представляет собой результат сопоставления строки с образцом. Операция $ENV{'USER'}=~mvz проверяет, содержится ли в строке $ENV{'USER'} регистрационное имя mvz (то есть зарегистрирован ли пользователь этой программы под указанным именем). Операция !~, напротив, осуществляет проверку на отсутствие соответствия.

Регулярные выражения в языке Perl

В языке Perl предусмотрена непосредственная трансляция регулярных выражений. Например, для распознавания регулярного выражения[1] a^+b^+ служит следующий сценарий Perl:

```
#!/usr/bin/perl   #сообщение операционной системе о том, что это сценарий на языке Perl
$_= <STDIN>;      #считывание введенной строки в специальную переменную $_
if (/^a+b+$/){print "yes\n";}
        else {print "no\n";}
```

Образец /X/ сравнивается с введенной пользователем строкой[2]. Проверка строки на соответствие регулярному выражению a^+b^+ будет успешной, если строка содержит цепочку символов, удовлетворяющую заданному регулярному выражению, начиная с первого символа (^) в строке и заканчивая последним ($). Если бы нужно было просто определить, содержит ли введенная строка *внутри* себя цепочку символов, удовлетворяющих регулярному выражению a^+b^+, то мы бы использовали образец /a+b+/. (Дополнительную информацию об операциях с регулярными выражениями можно найти в приложении, раздел П.9.)

[1] Это регулярное выражение задает цепочку символов, начинающуюся с одного или более символов a, за которыми обязательно следует один или более символов b. — *Примеч. науч. ред.*

[2] В Perl образец, заданный регулярным выражением, по умолчанию сопоставляется с содержимым специальной переменной $_. — *Примеч. науч. ред.*

Perl легко позволяет проверять соответствие строк заданному образцу и осуществлять подстановку новых значений:

```
$_ = 'aaabbbb';              #присваиваем встроенной переменной $_ значение 'aaabbbb'
if (s/(a+)/c/){print "$_ and $1\n";}
      else {print "fail\n"};
```

В этом примере s/X/Y/ означает следующую операцию: найти в переменной $_ цепочку символов, соответствующую образцу X, и заменить строкой Y. Образец a+ соответствует строке, содержащей один и более символов a, в нашем примере в исходной строке ему соответствует цепочка aaa, и поэтому эта часть строки заменяется символом c. Из-за того что образец заключен в скобки, переменной $1 присваивается значение цепочки, соответствующей образцу, поэтому в итоге будет напечатано "cbbbb and aaa", то есть будет представлен результат замены в соответствии с образцом и часть исходной строки, соответствующей заданному образцу.

3.3.4. Автоматы с магазинной памятью

В предыдущем разделе мы обсуждали конечные автоматы и установили, что те языки, которые допускаются такими автоматами, эквивалентны языкам, построенным при помощи регулярных грамматик. Таким образом, между регулярными грамматиками и конечными автоматами имеется определенное соответствие. Аналогично мы используем НФБ-грамматики для генерации цепочек некоторого языка и можем использовать абстрактную машину, распознающую, принадлежит ли заданная цепочка этому языку. В этом случае мы можем создать машину, называемую *автоматом с магазинной памятью*, или *МП-автоматом*, эквивалентную уже знакомым нам НФБ-грамматикам.

МП-автомат (Pushdown Automation, PDA) — это абстрактная машина, похожая на конечный автомат. МП-автомат также имеет конечный набор состояний, но, кроме того, в нем имеется стек (магазин). В МП-автомате осуществляются следующие переходы (такты):

1) считываются введенный символ и верхний символ в стеке;
2) в зависимости от этих двух значений состояние автомата меняется, и в стек записывается ноль или более символов;
3) цепочка считается допустимой, если стек становится пустым (может быть реализован и другой вариант: строка допускается, если МП-автомат достигает своего конечного состояния. Можно показать, что эти два варианта эквивалентны).

Легко заметить, что такие МП-автоматы обладают большими возможностями, чем КА. МП-автоматы могут распознавать цепочки типа $a^n b^n$, которые не распознаются при помощи КА. Нужно только поместить все символы a в стек и затем извлекать их оттуда по одному по мере того, как вводятся символы b. Если после ввода последнего b стек окажется пустым, то такая строка допускается МП-автоматом.

Менее очевидно то, что языки, допускаемые МП-автоматами, эквивалентны контекстно-свободным языкам. Однако рассмотрим процесс левостороннего вы-

вода цепочки символов[1]. В таком случае сентенциальную форму можно сохранить в стеке. Действия МП-автомата таковы.

1. Если верхним символом в стеке является терминальный, то он сравнивается с очередным введенным символом и в случае совпадения удаляется из стека. Несовпадение символов означает ошибку.
2. Если верхним символом в стеке является нетерминальный символ X, то он заменяется некоторой строкой α, где α — правая часть правила $X \to \alpha$.

Такой МП-автомат моделирует левосторонний вывод для некоторой контекстно-свободной грамматики. Приведенная конструкция фактически формирует недетерминированный МП-автомат, эквивалентный соответствующей НФБ-грамматике. На втором шаге работы нашего автомата можно использовать более одного правила вида $X \to \alpha$, и не всегда ясно, какое из них следует выбрать. *Недетерминированный МП-автомат* определяется аналогично недетерминированному конечному автомату. Цепочка допускается подобным автоматом, если существует необходимая для этого последовательность переходов.

Как соотносятся друг с другом детерминированный и недетерминированный МП-автоматы, если сравнивать их с аналогичными конечными автоматами? В данном случае существует отличие. Рассмотрим набор палиндромов, то есть строк, которые одинаково читаются в обоих направлениях, задаваемых следующей грамматикой:

```
S::= 0S0|1S1|2
```

При помощи детерминированного МП-автомата мы можем распознать такие строки следующим образом:

1) при чтении все символы 0 и 1 помещаем в стек;
2) при считывании символа 2 автомат переходит в новое состояние;
3) сравниваем каждый новый введенный символ с верхним символом в стеке и удаляем его из стека.

Теперь рассмотрим следующее множество палиндромов:

```
S::= 0S0|1S1|0|1
```

В этом случае мы никогда не будем знать, где находится середина строки. Для распознавания таких палиндромов автомату придется ее найти. Например, в случае палиндрома 011010110 последовательность действий МП-автомата могла бы быть такой:

Стек	Середина	Стек сравнивается с
	0	11010110
0	1	1010110

продолжение ⇗

[1] *Вывод* в контекстно-свободной грамматике называется левосторонним, если ее правила применяются к самому левому вхождению нетерминального символа каждой цепочки вывода. Приведенное доказательство эквивалентности языков, допускаемых МП-автоматами, и контекстно-свободных языков основывается на том, что в произвольной контекстно-свободной грамматике можно построить эквивалентный любому выводу левосторонний вывод. — *Примеч. науч. ред.*

Продолжение таблицы

Стек	Середина	Стек сравнивается с
01	1	010110
011	0	10110
0110	1	0110
01101	0	110
011010	1	10
0110101	1	0
01101011	0	

Только пятый шаг — когда автомат определяет, что первой половиной палиндрома является 0110,— заканчивается успешно. Если некоторая последовательность шагов определения середины цепочки приводит к ее полному грамматическому разбору, то такая цепочка считается допустимой с точки зрения данной грамматики.

3.3.5. Общие алгоритмы грамматического разбора

После появления работы Хомского со временем стало ясно, что каждый тип формальной грамматики тесно связан с определенным автоматом — простой абстрактной машиной, которая обычно определяется как машина, считывающая с входной рабочей ленты последовательность символов, хранящихся в ее ячейках, и производящая выходную ленту, в ячейках которой содержатся символы другой последовательности. К сожалению, здесь возникает проблема. Поскольку НФБ-грамматика может оказаться неоднозначной, соответствующий ей автомат должен быть *недетерминированным* (то есть может быть несколько вариантов перехода, из которых автомат должен выбрать наиболее подходящий).

Недетерминированный МП-автомат может распознавать любые цепочки, порождаемые контекстно-свободной грамматикой, используя стратегию предположений (guessing strategy). Однако для целей программирования и трансляции программ требуются более ограниченные в своих действиях автоматы (*детерминированные* автоматы), которым не приходится заниматься предположениями.

Хотя регулярной грамматике всегда соответствует детерминированный аппарат, в случае НФБ-грамматики это верно, только если грамматика является однозначной и выполнены некоторые другие требования. Для однозначных НФБ-грамматик были разработаны методы непосредственного грамматического разбора. Одним из самых первых был метод рекурсивного спуска. Большим шагом вперед стало открытие Кнутом класса так называемых LR-грамматик (left to right parsing algorithms — грамматика с ограниченным левым контекстом), которые описывают все НФБ-грамматики, распознаваемые детерминированным МП-автоматом. Класс LR(1)-грамматик включает все такие грамматики, в которых требуется знать один только следующий символ, чтобы в процессе грамматического разбора принять правильное решение. SLR- (Simple LR — простая LR) и LALR-

(Lookahead LR — LR с «заглядыванием вперед») грамматики являются подклассами класса LR-грамматик, которые приводят к эффективным методам грамматического разбора. Альтернативой является нисходящий метод LL-грамматики, обобщающий метод рекурсивного спуска. Большинство современных языков программирования разработано на основе одной из грамматик, SLR, LR или LL, и для них можно использовать инструменты генерации распознавателей типа YACC для автоматического создания распознавателя, соответствующего заданной грамматике.

Детерминированные МП-автоматы эквивалентны LR(k)-грамматикам и имеют значение при разработке практических компиляторов для языков программирования. В большинстве языков, построенных на основе грамматик, для определения синтаксиса используется некоторая LR(k)-грамматика. Подробное изучение LR(k)-грамматик выходит за рамки этой книги, но в разделе 3.4 мы приведем краткое описание грамматического разбора на основе метода рекурсивного спуска как пример грамматического разбора контекстно-свободных языков.

3.4. Грамматический разбор на основе метода рекурсивного спуска

В задачи этой книги, как уже было сказано, не входит изучение полного спектра методов грамматического разбора. Тем не менее метод рекурсивного спуска является относительно простым в описании и реализации, и на его примере можно показать, как связано формальное описание языка программирования с возможностью генерации выполняемого кода для программ на этом языке.

Напомним, что мы всегда можем переписать грамматику, используя расширенную НФБ-нотацию. Например, для синтаксиса оператора присваивания, представленного в табл. 3.3, арифметическое выражение описывается следующим образом:

```
<арифметическое выражение> ::= <терм>{[+ | -] <терм>}*
```

Это означает, что в первую очередь должна распознаваться синтаксическая категория <терм>. Если следующим символом окажется «+» или «–», то за ним должна последовать другая синтаксическая категория, <терм>. Предположим, что переменная nextchar всегда содержит первый символ соответствующего нетерминального символа, а функция getchar считывает символ. Тогда мы можем непосредственно переписать прежнее правило, записанное в расширенной НФБ-нотации, в виде следующей рекурсивной процедуры:

```
procedure Expression;
begin
   Term;                          /*вызов процедуры Term для поиска первого терма*/
      while ((next char= '+') or (nextchar= '-')) do
      begin
         nextchar := getchar;     /*Пропуск символа операции*/
         Term
      end
end
```

Листинг 3.1. Распознаватель на основе метода рекурсивного спуска для арифметических операторов[1]

```
procedure AssignStmt
        begin
        Variable:
        if nextchar <> '=' then Error
            else begin
            nextchar := getchar:
            Expression
            end
    end:
procedure Expression
        begin
        Term:
        while ((nextchar = '+')or(nextchar= '-')) do
            begin
            nextchar := getchar;
            Term
            end
    end:
procedure Term
        begin
        Primary:
        while ((nextchar = 'x')or(nextchar= '/')) do
            begin
            nextchar := getchar:
            Primary
            end
    end:
procedure Primary
        begin if nextchar = letter then Variable
        else if nextchar = digit then Number
        else if nextchar = '(' then
            begin
            nextchar := getchar:
            Expression:
            if nextchar = ')' then
                nextchar := getchar
                else Error/*Пропущено ')' */
            end
        else Error/*Пропущено '(' */
        end:
procedure Variable:
        begin Identifier:
        if nextchar = '[' then
        begin
            nextchar := getchar;
            SubList:
            if nextchar = ']' then
            nextchar := getchar
            esle Error/*Пропущено ']' */
        end
end;
```

[1] В листинге 3.1 Identifier и Number — это функции, предназначенные для чтения идентификаторов и чисел соответственно при помощи лексического распознавателя на основе КА.

```
procedure SubList:
      begin Expression:
      while nextchar = '.' do:
         begin
         nextchar := getchar;
         Expression
         end
end;
```

В листинге 3.1 полностью представлен распознаватель на основе метода рекурсивного спуска для операторов присваивания, задаваемых грамматикой, представленной в табл. 3.3. Чтобы закончить разговор о грамматическом разборе, следует указать, что постфиксная запись выражения $term_1$ + $term_2$ выглядит следующим образом: $term_1$ $term_2$ + (впрочем, мы еще вернемся к этому в разделе 8.2.1). Как будет показано далее, преобразование операторов исходной программы в постфиксную запись позволяет использовать удобную в применении стратегию вычисления выражений. Если у нас имеются процедуры для распознавания арифметических выражений, то не составляет труда произвести постфиксную запись этих выражений. Предположим, что каждая процедура производит постфиксную запись для собственных подвыражений, используя процедуру output. Постфиксная запись для арифметического выражения в процедуре Expression может быть получена следующим образом:

```
procedure Expression:
begin var PlusType: char:
   Term:  /*вызов процедуры Term для поиска первого терма*/
      while ((next char= '+') or (nextchar= '-')) do
      begin
            PlusType :=nextchar:
            nextchar := getchar:
            Term: output(Plustype)
      end
end
```

Все остальные процедуры можно модифицировать аналогичным образом. Постфиксная запись для оператора переменная = выражение выглядит как переменная выражение =, для выражения $множитель_1 \times множитель_2$ — соответственно $множитель_1$ $множитель_2$ $\times$ и т. д.

3.5. Обзор языка Pascal

История создания. Pascal разрабатывался с 1968 по 1970 г. Николаусом Виртом (Niklaus Wirth). Цель заключалась в том, чтобы создать язык, лишенный многочисленных недостатков ALGOL. Pascal был назван в честь французского математика Блеза Паскаля, который еще в 1642 г. изобрел цифровой калькулятор. С конца 70-х до конца 80-х гг. этот язык доминировал среди языков, используемых на начальном этапе обучения программированию; позже его заменили C и C++, а затем Java.

ALGOL 60 был первой попыткой создания языка на основе формального описания, однако его реализация оказалась сложной. В частности, оказалось достаточно трудно реализовать передачу параметров по имени, хотя это довольно эле-

гантный механизм. В языке ALGOL 60 не были определены операторы ввода-вывода, поскольку в то время считалось, что они зависят от реализации, да и собственную статическую память также трудно было реализовать. Помимо того, в 60-х гг. были разработаны новые практические решения, например типы данных и структурное программирование. Языки типа FORTRAN были популярны благодаря своей эффективности при выполнении программ, несмотря на отсутствие элегантности.

В 1965 г., во время работы в Стенфордском университете (Standford University), Вирт разработал новую, расширенную версию ALGOL 60 для компьютеров серии IBM 360, в которую вошло определение указателей и структур данных. Этот язык, известный как ALGOL W, использовался в нескольких университетах, но его реализация ограничивалась только компьютерами IBM 360. Для выполнения программ на этом языке требовался значительный по размерам пакет программ поддержки обработки строк, вещественных чисел двойной точности и других сложных типов данных. Таким образом, ALGOL W в качестве системного языка программирования оказался малоэффективным.

В 1968 г. Вирт вернулся в Швейцарию и начал работу над преемником ALGOL W — языком, который мог бы компилироваться за один проход. Для создания исходного компилятора был использован алгоритм рекурсивного спуска. Этот компилятор выполнялся на компьютере Control Data. Также был разработан широко известный теперь интерпретатор P-кода. Компилятор языка Pascal сначала транслировал исходную программу в программу на языке гипотетической машины со стековой архитектурой. Благодаря такой своей организации Pascal легко переносился на компьютеры других систем. Компилятор Pascal был написан на одноименном языке. Все, что требовалось для перехода в другую систему, — это переписать соответствующим образом интерпретатор P-кода.

Появившийся в 1970 г. Pascal начал завоевывать признание. В 1983 г. был разработан американский стандарт языка (IEEE 770/ ANSI X3.97 [70]), а вскоре был разработан стандарт ISO (ISO 7185).

Краткий обзор языка. Структура программ на языке Pascal напоминает программы на C. Тем не менее в Pascal предусмотрена возможность описания внутренних локальных процедур и создания вложенной иерархии имен. Программа на Pascal представляет собой единый программный блок, в котором содержатся определения используемых подпрограмм.

В Pascal имеется достаточно широкий набор простых и структурированных типов данных: целые и вещественные числа, символьные данные, перечисления, логические (булевы) значения, массивы, записи, последовательные файлы и ограниченный тип множеств. Оператор type позволяет программисту определять новые типы данных, хотя не обеспечивает группирование и инкапсуляцию определения нового типа данных с набором подпрограмм, обеспечивающих выполнение основных операций над объектами данных этого нового типа. Кроме того, указатель и операция создания новых объектов данных любого типа позволяют программисту конструировать новые объекты связанных данных непосредственно во время выполнения программы.

Подпрограммы принимают форму функций (если они возвращают одно какое-либо значение) или процедур (если их действие сводится к модификации пере-

данных параметров или глобальных переменных). Операторы управления последовательностью действий базируются на конструкциях структурного программирования: составных операторах, условных операторах и операторах выбора (case), а также трех видах операторов цикла. В Pascal имеется также оператор goto, который редко используется и без которого практически всегда можно обойтись. Вызов подпрограмм и возвращение значений осуществляется с помощью обычной рекурсивной структуры вызова-возврата.

Поскольку Pascal имеет блочную структуру, большая часть структур управления данными для ссылок на переменные использует стандартные статические правила определения области видимости и характеристику вложенности блока в самой программе. Параметры могут передаваться по ссылке или по значению.

Pascal можно эффективно реализовать на обычном аппаратном компьютере. Идеология языка включает только те языковые свойства, для которых существуют хорошо изученные и эффективные методики реализации. Во время трансляции почти для всех операций возможен статический контроль типов, так что необходимость в динамическом контроле минимальна, но при этом обеспечивается полная безопасность выполнения. Обычно программа транслируется в выполняемый машинный код, но в некоторых реализациях Pascal результатом трансляции является виртуальный машинный код, который затем интерпретируется и выполняется при помощи некоторого программно-моделируемого интерпретатора.

Во время выполнения программ на Pascal центральный стек используется для записей активации подпрограмм, область динамически распределяемой памяти отводится под объекты данных, созданных для прямого манипулирования с помощью переменных-указателей, а область статически распределяемой памяти используется для хранения сегментов кода подпрограмм и вспомогательных подпрограмм из библиотеки поддержки выполнения. Из вспомогательных подпрограмм нужны в основном стандартные программы ввода-вывода для последовательных файлов и процедуры для управления ресурсами памяти.

Хотя Pascal в целом очень удобный и полезный язык, у него есть свои недостатки, перечень которых приведен ниже.

1. В определении этого языка имеется некоторое противоречие между идеологией самого языка и его реализацией. Например, конструкция forward нужна только для того, чтобы компиляция могла выполняться в один проход, — это следствие представлений о том, что таким образом достигается максимальная эффективность компиляции. Но это не всегда верно. Например, компилятор PL/C для языка PL/I совершал три прохода и вместе с тем являлся одним из самых эффективных среди наиболее распространенных компиляторов своего времени [32]. Кроме того, в настоящее время при использовании недорогих быстродействующих компьютеров скорость компиляции не имеет большого значения.

2. Возможно, самой главной слабостью языка Pascal является то, что массивы рассматриваются как отдельные типы, а не как агрегация различных объектов одного типа. Это приводит к тому, что, например, array[1..10] of integer и array[1..20] of integer представляют собой *разные типы* данных. В результате алгоритмы обработки массивов усложняются, поскольку массивы различных размеров невозможно передать общей подпрограмме (например,

подпрограмме перемножения матриц). Строки реализованы как массивы символов, что также затрудняет их обработку в случае строк различной длины.

3. Синтаксис определения процедуры в Pascal выглядит следующим образом:

```
заголовок процедуры
локальные переменные
локальные параметры
begin тело_процедуры end
```

   Поскольку в программе может содержаться большое количество вложенных локальных процедур, то определение локальной переменной, которая используется в какой-либо процедуре, оказывается (синтаксически) сильно отдаленным от места ее использования в теле подпрограммы. Это приводит к затруднениям при создании документации и чтении больших программ на Pascal.

4. Возможности, предоставляемые языком, должны выполняться не с помощью *пропуска* некоторой информации, а *явным указанием* этой информации. В Pascal передача параметров нарушает это правило. Все параметры в Pascal передаются по значению, если только в списке параметров не указан явным образом атрибут var, который означает, что соответствующий параметр должен передаваться по ссылке. Многие начинающие программисты (в том числе один из авторов этой книги) часами рассматривали листинги программ, стараясь обнаружить ошибку, связанную с пропуском ключевого слова var.

5. Pascal был реализован таким образом, что компиляция программы представляла собой единый процесс, то есть не была предусмотрена возможность компилировать отдельные программные модули. В большинстве реализаций, однако, эту проблему удалось решить: было принято соглашение, что допускаются дополнительные внешние процедуры, аналогичные заголовочным файлам с расширением .h в языке С. Но такая нестандартная реализация ограничивает возможность перенесения программ на Pascal на другие машины.

6. Хотя в Pascal допускается определение новых типов данных для поддержки абстракций, в нем фактически не предусмотрена возможность инкапсуляции и сокрытия информации. (Это замечание является скорее не критикой данного языка, а комментарием, касающимся общего уровня развития программирования в 1970 г., когда создавался Pascal.)

3.6. Рекомендуемая литература

Имеется довольно обширная литература по синтаксису и трансляции языков программирования. В книгах Ахо, Сетхи и Ульмана [8], а также Фишера и Лебланка [41] достаточно подробно исследуется процесс трансляции программ. В главе 10 книги Берманна [17] эта же тема представлена в более простом, обзорном виде. Идеология диагностирующих компиляторов описана в документации языков

CORC [31] и PL/I [32], разработанных в Корнельском университете. Исправление синтаксических ошибок обсуждается в работе Моргана [84].

Практические соображения по разработке синтаксиса языков программирования также нашли свое отражение в печатных работах. В частности, многие вопросы по этой тематике освещаются в соответствующих работах Саммета [95]. В документации по таким языкам программирования, как Ada [58] дается обоснование решений, принятых при разработке этих языков.

3.7. Задачи и упражнения

1. Рассмотрим следующие правила НФБ-грамматики:

```
<pop> ::= [<bop>. <pop>]|<bop>
<bop> ::= <boop>|(<pop>)
<boop>::= x|y|z
```

   Для каждой из перечисленных ниже строк укажите все синтаксические категории, к которым они относятся (если такие существуют):

   - c
   - (x)
   - [y]
   - ([x.y])
   - [(x).y]
   - [(x).[y.x]]

2. Предложите однозначную грамматику для языка, определяемого следующим правилом:

   S → SS|(S)|()

3. Покажите, что любое заданное дерево грамматического разбора может быть результатом нескольких выводов (например, покажите, что существует несколько выводов, генерирующих дерево, показанное на рис. 3.4). Что можно сказать о дереве грамматического разбора, если грамматика является однозначной? Неоднозначной?

4. Определим *левый вывод* как вывод, в котором правило применяется к самому левому из всех нетерминальных символов в каждой сентенциальной форме. Покажите, что грамматика является неоднозначной в том и только том случае, если в языке существует цепочка, имеющая два различных левых вывода.

5. Напишите НФБ-грамматику для языка, состоящего из всех двоичных чисел, которые содержат по меньшей мере три последовательно идущие единицы. (Язык включает, например, цепочки 011101011, 000011110100 и 1111110, но не 0101011.)

6. Синтаксис *обезьяньего* языка очень прост, но только обезьяны умеют говорить на нем, не делая ошибок. Алфавит состоит из {a,b,d,#}, причем # используется в качестве пробела. Грамматика задана следующим образом:

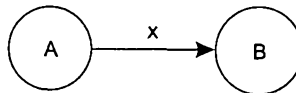
```
<остановка> ::= b|d
<звук> ::= <остановка>a
<слог> ::= <звук>|<звук><остановка>|a<звук>|a<остановка>
```

```
<слово> ::= <слог>|<слог><слово><слог>
<предложение> ::= <слово>|<предложение>#<слово>.
```

Кто из перечисленных ниже ораторов — секретный агент, замаскированный под обезьяну?

```
Горилла: ba#ababadada#bad#dabbada
Шимпанзе: abdabaadab#ada
Бабуин: dad#ad#abaadad#badadbaad
```

7. Приведите регулярные выражения:
   - для всех бинарных цепочек, заканчивающихся на 01;
   - для целых десятичных чисел, кратных 5;
   - для идентификаторов языка C;
   - для бинарных цепочек, состоящих либо из нечетного количества единиц, либо нечетного количества нулей.
8. Покажите, что любой КА может быть представлен регулярной грамматикой и любая регулярная грамматика может быть распознана КА. Чтобы решить эту задачу, следует связать каждый нетерминальный символ в грамматике с состоянием КА. Например, схема, представленная на рис. 3.10, соответствует правилу A → xB. (Как вы обработаете конечные состояния?)



Переход конечного автомата

A ——→ xB

Правило регулярной грамматики

Рис. 3.10. Связь грамматики и КА

9. Постройте конечный автомат и регулярную грамматику для следующих цепочек:
   - (ab∨ba)*∨(ab)*;
   - (11*)*(110∨01);
   - всех цепочек, состоящих из 0 и 1 и содержащих подцепочку 010;
   - всех цепочек, состоящих из 0 и 1 и не содержащих подцепочку 010.
10. Напишите расширенную НФБ-грамматику для языков, определенных конечными автоматами, изображенным на рис. 3.6 и 3.7.
11. Постройте дерево грамматического разбора для следующих операторов присваивания, используя НФБ-грамматику, представленную в табл. 3.1:
   - A[2] := B +1
   - A[I,J] := A[J,I]
   - X:=U - V × W + X / Y
   - P:= U / (V / (W / X))

12. Определение недетерминированного КА включает в себя определение детерминированного КА как особый случай. Предположим, однако, что у нас имеется недетерминированный конечный автомат N. Покажите, что тогда существует детерминированный автомат D, который принимает тот же самый язык. Это означает, что отсутствие детерминированности у КА не увеличивает вычислительную мощность автомата N. Для того чтобы показать это, рассмотрим, скольких состояний в N мы можем достичь из заданного состояния и заданного входного символа. Назовем это множество состояний $Состояния_i$. Для автомата N, содержащего n состояний, таких подмножеств будет 2^n. Это определяет множество состояний нашего детерминированного КА.
13. Пусть S — это регулярное множество (то есть множество, распознаваемое конечным автоматом). Покажите, что S^R (обращенное множество S, то есть множество цепочек из S, каждая из которых записана в обратном порядке) также является регулярным множеством.
14. Пусть R — регулярное множество. Пусть $R^{1/2}$ — это множество, составленное из левых «половинок» элементов R, то есть если элемент w принадлежит R и длина этого элемента 2k, то первые k символов из w являются элементом $R^{1/2}$. Покажите, что $R^{1/2}$ также является регулярным множеством. (Подсказка: решите сначала предыдущую задачу для S^R.)

Глава 4. Моделирование свойств языка

На начальном этапе развития компьютерных технологий среди профессионалов в этой области господствовало мнение, что формальный синтаксис (например, НФБ-грамматика) достаточен для описания атрибутов языка программирования и, следовательно, для полного и однозначного описания поведения программы. Но оказалось, что это не так, и в разделе 3.3.2 были приведены некоторые примеры, демонстрирующие недостаточность НФБ-грамматики для точного описания поведения программы.

НФБ-грамматика является прекрасным инструментом для ответа на вопрос: как должна выглядеть программа на конкретном языке? В этой главе[1] мы рассмотрим несколько формальных теорий, которые расширяют НФБ-грамматику и позволяют ответить на вопрос: что делает данная программа? Для этого мы введем следующие формальные модели.

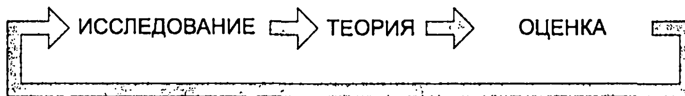
1. *Формальные грамматики.* НФБ-грамматика и регулярная грамматика — всего лишь два примера более сложной структуры, разработанной в 50-е гг. прошлого столетия профессором Массачусетсского технологического института (MIT) Ноамом Хомским. Мы познакомим с иерархией грамматик Хомского и дадим краткое описание их свойств. Мы покажем, что, к сожалению, к интересующим нас проблемам языков программирования имеют отношение только уже исследованные нами НФБ-грамматика и регулярная грамматика. Тем не менее другие классы грамматик обладают важными свойствами, отражающими природу вычислений, и в этой главе мы познакомим читателя с концепциями неразрешимости и алгоритмической сложности.
2. *Семантика языка.* Для разработки моделей языков программирования помимо формальной грамматики используются и другие подходы. Одним из первых среди них стала атрибутивная грамматика, предложенная профессором Стенфордского университета (Standford University) Дональдом Кнутом (Donald Knuth). В этой модели к обычной НФБ-грамматике, описывающей язык, добавлена некоторая семантическая информация. Более формальной моделью является денотационная семантика, которая записывает программу в виде математической функции.

[1] Эта глава несколько сложнее остальных глав этой книги, поэтому она предназначена только для тех читателей, которым требуется дополнительная информация. Остальные могут пропустить ее, за исключением раздела 4.2.3, в котором приведен обзор языка ML.

3. *Верификация программы.* Третий подход заключается в верификации программы. Целью здесь является скорее не формальное описание программы, а демонстрация эквивалентности программы и некоторого другого способа записи — например, логики предикатов. Например, если требуется написать программу, вычисляющую куб заданного числа, то в данном способе предлагается доказать, что программа и уравнение $x = y^3$, где x — это выходное значение, а y — введенное число, ведут себя одинаково. В данном случае нас интересует не столько то, что представляет собой программа, сколько правильность ее поведения. Исходно эта технология была разработана в 1960 г. Робертом Флойдом (Robert Floyd) и Т. Хоором (Tony Hoare). Хотя ее трудно реализовать, она имеет достаточно большое значение в определенных приложениях, для которых важнейшим требованием является правильность программы (например, в приложениях, контролирующих работу ядерных реакторов).

4.1. Формальные свойства языков

Современная цивилизация основана на взаимодействии науки и техники, целью которого является объяснение устройства окружающего мира и развитие технологий, способствующих дальнейшему прогрессу. Общая модель этого взаимодействия может быть представлена следующей схемой:



Мы проводим исследования и получаем некоторое знание (например, мы узнаем, что в программах неизбежны ошибки и, вообще, программирование — сложное занятие), развиваем теории разработки языков программирования (например, структурное программирование, НФБ-грамматики, абстракции данных), а затем оцениваем действенность этих теорий (например, используем их для создания языков). Это приводит к дальнейшему развитию теории (например, развитие концепции объектно-ориентированных классов в результате усовершенствования понятия абстракций данных или методов разбора на основе SLR[1]- и LALR[1]-грамматик в результате развития метода разбора на основе LR[1]-грамматики) и оказывает благотворное влияние на весь процесс в целом. В предыдущих главах мы уже обсудили некоторые из общепринятых методов разработки языков программирования. В данном разделе мы продолжим этот разговор и коротко расскажем о некоторых других теоретических моделях, которые влияют на разработку языков программирования.

Разработка и реализация большинства ранних языков программирования основывались на практических соображениях. Вопрос стоял таким образом: как извлечь пользу из использования некоторого примитивного устройства, принадлежащего к аппаратной части компьютера, при создании больших программ, необходимых для решения таких задач, как конструирование самолета, или анализ

данных радара, или управление ядерной реакцией? Возникновение многих ранних языков было обусловлено необходимостью решения подобных задач, при этом роль теории была незначительна. Многочисленные проблемы, встретившиеся разработчикам на этом пути, равно как и удачные решения, привели к возникновению ранних формальных моделей синтаксиса и семантики языков программирования, что, в свою очередь, способствовало созданию более совершенных языков программирования. В новых языках по-прежнему встречались многочисленные недоработки и упущения как в стадии разработки, так и при их реализации. Совершенствование теоретических моделей вело к прогрессу в разработке языков программирования.

Теоретическая модель может быть *концептуальной* (то есть качественной), в таком случае она описывает язык в терминах лежащих в ее основе базовых понятий и не претендует на формальное математическое описание этих понятий. Именно в этом смысле в предыдущих главах была построена теоретическая модель базовых понятий, лежащих в основе разработки и реализации языков программирования. С другой стороны, теоретическая модель может быть *формальной* (то есть количественной), в которой исследуемое явление описывается в терминах точной математической модели, которую можно изучать, анализировать и преобразовывать, используя доступный математический аппарат. Теоретические модели, рассматриваемые в этой главе,— это формальные модели. Их описания, которые вы здесь найдете, не претендуют на полноту, но содержат достаточно информации для понимания проблемы и возможных путей ее решения.

4.1.1. Иерархия грамматик Хомского

Как было сказано в разделе 3.3.1, НФБ-грамматики оказались очень полезными при описании синтаксиса языков программирования. Однако эти НФБ-грамматики (или контекстно-свободные грамматики) — всего лишь один из элементов класса грамматик, описанных Ноамом Хомским в 1959 г. [27]. Мы приводим здесь краткое описание первоначально предложенной им модели грамматик.

Типы грамматик

В разделе 3.3.1 мы представили базовый синтаксис правил, или *продукций*, НФБ-грамматики. *Грамматика* определяется как совокупность множества нетерминальных символов, множества терминальных символов, начального символа (один из нетерминальных) и множества правил. Классы грамматик различаются между собой в зависимости от имеющегося набора допустимых правил.

В случае НФБ-грамматики *язык* определяется просто как набор конечных последовательностей (цепочек) символов некоторого произвольного алфавита, выведенных из одного начального символа. Алфавит — это набор символов, которые используются при написании программ, и каждая законченная программа представляет собой их последовательность. Мы можем говорить о множестве цепочек, *генерируемых* грамматикой (то есть цепочек терминальных символов, выведенных из начального символа), или, наоборот, мы можем сказать, что грамматика *распознает* цепочки (то есть по некоторой цепочке всегда можно построить дерево синтаксического разбора и вернуться к исходному начальному символу).

Будем называть языком типа n язык, который генерируется грамматикой типа n, если нет грамматики типа $n + 1$, которая также генерирует этот язык. Из этого определения следует, что любая грамматика типа n является также и грамматикой типа $n - 1$.

Грамматики типа 3 — это просто регулярные грамматики, определяющие автоматные языки, моделирующие лексические единицы языка. Грамматики типа 2 — это знакомые нам НФБ-грамматики. Грамматики типа 1 и 0 не имеют большого практического значения для моделирования языков программирования, но в области теоретических исследований они играют существенную роль. Обзору каждого из упомянутых типов посвящены следующие несколько разделов.

Регулярные грамматики (тип 3)

Как было сказано ранее, конечный автомат и регулярные грамматики обеспечивают модель для конструирования лексического анализатора в трансляторе некоторого языка программирования (см. раздел 3.3.2). Свойства регулярных грамматик таковы:

- Большинство свойств такой грамматики разрешимы. (Это означает, что можно получить ответы, например, на такие вопросы: генерирует ли данная грамматика любые цепочки? Сгенерирована ли этой грамматикой заданная цепочка символов языка? Ограничено ли количество цепочек в языке?)
- Для любой конечной последовательности α и целого n регулярная грамматика может генерировать строки вида α^n. Это означает, что регулярная грамматика может распознавать любое количество образцов конечной длины.
- С помощью регулярной грамматики можно реализовать счет до любого конечного целого числа. Например, вы можете распознать цепочку $\{a^n \mid n = 147\}$, если построите КА, в котором будет как минимум 148 состояний (рис. 4.1). Ввод первых 146 символов не приведет КА в конечное состояние, тогда как очередной ввод символа под номером 147 будет принят. Ни один КА с числом состояний ровно 148 не сможет надежно принять цепочку символов длиной больше 147.

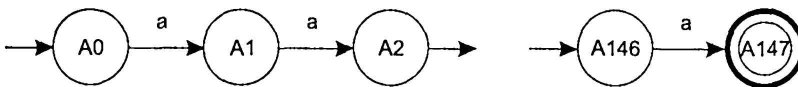


Рис. 4.1. Использование регулярной грамматики в счетчике

- Грамматики такого типа часто используются в сканерах (лексических анализаторах) компиляторов для распознавания отдельных лексем (идентификаторов, литералов и строк) или ключевых слов данного языка (например, if, begin, while).

В качестве примера можно рассмотреть регулярную грамматику, генерирующую идентификаторы Pascal (которые, как вы помните, состоят из букв и цифр, причем на первом месте стоит буква, то есть имеют вид буква{буква∨цифра}*). Она задается следующим образом:

```
Идентификатор → aX|...|zX|a|...|z
X → aX|...|zX|0X|...|9X|a|...|z|0|...|9
```

Контекстно-свободные грамматики (тип 2)

Правила таких грамматик являются правилами уже известных нам НФБ-грамматик. Они записываются в форме $X \to \alpha$, где под α понимается любая последовательность терминальных или нетерминальных символов.

Эти грамматики характеризуются следующими свойствами.

- Многие свойства такой грамматики являются разрешимыми. (Это означает, что можно получить ответы, например, на такие вопросы: генерирует ли данная грамматика какие-либо цепочки? Сгенерирована ли этой грамматикой данная цепочка языка? Является ли язык, порождаемый грамматикой, пустым?)
- Такие грамматики можно использовать для подсчета вхождения в цепочку двух символов и последующего сравнения. То есть они характеризуются цепочками вида $a^n c b^n$ для любого n.
- Контекстно-свободная грамматика может быть «реализована» при помощи стеков. Для распознавания цепочки $a^n c b^n$ из предыдущего пункта можно занести в стек цепочку a^n, затем проигнорировать c и сравнить содержимое стека с цепочкой b^n, чтобы удостовериться в одинаковой длине этих двух цепочек.
- Как описано в главе 3, эти грамматики можно использовать для автоматического построения дерева грамматического разбора.
- Грамматики типа 2 и типа 3 по большей части более не представляют интереса и не являются объектом исследований. Судя по всему, все важные свойства этих грамматик уже изучены.

В качестве примера приведем следующую грамматику, определяющую обычное арифметическое выражение:

```
E → E + T | T
T → T * P | P
P → i | (E)
```

Контекстно-зависимые грамматики (тип 1)

Такой тип грамматики характеризуется правилами вида:

$$\alpha \to \beta$$

где α — это любая цепочка, состоящая из нетерминальных символов, β — любая цепочка, состоящая из терминальных и нетерминальных символов, и количество символов в α меньше или равно количеству символов в β.

Ниже перечислены некоторые свойства контекстно-зависимых грамматик:

- Все цепочки, последовательно выводимые из начального символа, имеют длину ***не меньшую, чем предыдущая***, поскольку каждое правило должно оставлять длину цепочки неизменной или увеличивать ее[1].
- Контекстно-зависимые грамматики генерируют цепочки, для хранения которых требуется фиксированный объем памяти. Например, такие грамма-

---

[1] В связи с указанным свойством контекстно-зависимых грамматик их еще называют неукорачивающими грамматиками. — *Примеч. науч. ред.*

тики способны распознавать цепочки вида $a^n b^n c^n$, что не может сделать контекстно-свободная грамматика.

- Контекстно-зависимые грамматики обычно слишком сложны, чтобы быть практически полезными для моделирования языков программирования.
- Некоторые свойства контекстно-зависимых грамматик до сих пор не исследованы. В разделе 4.1.3 мы коснемся теоретической задачи доказательства НП-полноты, то есть проблемы эквивалентности детерминированных и недетерминированных контекстно-зависимых грамматик.

В качестве примера рассмотрим генерацию строки $xa^n b^n c^n$ при помощи контекстно-зависимой грамматики:

```
X → ABCX|Y
CB → BC
CA → AC
BA → AB
CCY → CYc
BCY → BYc
BBY → BYb
ABY → AYb
AAY → AYa
AY → xa
```

Грамматики с фразовой структурой (тип 0)

Этот тип грамматик характеризуется ничем не ограниченным набором правил вида $\alpha \to \beta$, где α — это любая строка нетерминальных символов, а β — любая строка, составленная из терминальных или нетерминальных символов[1].

Свойства этого типа грамматик следующие:

- Они могут использоваться для распознавания любой вычислимой функции. Например, можно создать грамматику (хотя это и не очень просто) для цепочки $a^n b^{f(n)}$, представляющей функцию f(n). По заданному n — числу элементов a, эта грамматика генерирует цепочку, содержащую f(n) элементов b.
- Большая часть свойств этих грамматик относится к *неразрешимым* (см. раздел 4.1.3). Это означает, что не существует процесса, с помощью которого можно было бы определить, выполняется ли данное свойство для всех грамматик данного типа (например, является ли множество цепочек данного языка пустым?). Отличие от контекстно-зависимых грамматик заключается в том, что у последних о многих свойствах просто ничего не известно — они могут быть истинными, ложными или быть неразрешимыми.

4.1.2. Неразрешимость

При обсуждении иерархии Хомского вы могли заметить, что по мере продвижения от грамматик типа 3 к грамматикам типа 0 усложнялись соответствующие язы-

[1] В англоязычной научной литературе в связи с отсутствием ограничений на набор правил грамматики их так и называют «unrestricted grammars» — «неограниченные грамматики». — *Примеч. науч. ред.*

ки. Мы знаем, что современные компьютеры работают очень быстро и обладают почти невероятными способностями в области решения задач. Поэтому закономерен следующий вопрос: существует ли какой-либо предел для вычислений с помощью компьютера?

Рассмотрим следующую практическую задачу. Вместо того чтобы тестировать программу, написанную, скажем, на языке C, нельзя ли написать некоторую другую программу, которая могла бы по описанию C-программы (например, по ее исходному файлу с текстом программы) определить, произойдет ли остановка во время выполнения последней? Такая тестирующая программа была бы чрезвычайно полезна, так как с ее помощью можно было бы избежать многочисленных проверок программ, которые со временем входят в бесконечные циклы.

Но если вы попробуете написать такую тестирующую программу, вы убедитесь, что это чрезвычайно сложная, практически невыполнимая задача. Дело не в том, что у вас для этого недостаточно знаний или способностей; написать такую программу не под силу никому. Это ограничение — одно из следствий той математической системы, которая используется для написания программ, что становится ясным при изучении языков типа 0. В этом разделе мы обсудим некоторые аспекты возникшей проблемы.

Машины Тьюринга

Когда мы программируем на некотором языке (назовем этот язык A), обычно достаточно очевидно, что эквивалентная программа может быть написана на языке B. Например, если вы пишете программу для составления платежной ведомости на языке COBOL, то такую же программу вы могли бы написать на языках C или FORTRAN; возможно, несколько сложнее было бы написать ее на LISP или ML. Существуют ли такие программы, которые могут быть написаны только на каком-то конкретном языке, то есть для которых невозможна эквивалентная замена программой, написанной на другом языке? Например, возможны ли программы на LISP или Prolog, для которых не существует эквивалентной программы на языке FORTRAN? Для того чтобы точнее сформулировать данный вопрос, нужно ввести понятие *универсального языка программирования*. Универсальный язык программирования — это такой язык, на котором можно запрограммировать любое вычисление. Тогда вопрос сводится к следующему: *являются ли все стандартные языки программирования универсальными*? Если нет, то какие типы программ не могут быть написаны ни на каком другом языке, кроме некоторого одного? Если да, то зачем нам нужно такое количество различных языков программирования? Может быть, следует выбрать из них какой-то один, наиболее простой, который позволит обойтись без всех остальных языков?

В первую очередь отметим, что этот вопрос можно переформулировать в терминах функций, вычисляемых программой. Эквивалентность некоторой программы P на языке A какой-либо другой программе Q, написанной на языке B, означает, что обе эти программы вычисляют одну и ту же функцию. Иначе говоря, в каждом конкретном случае обе программы получают одни и те же входные данные и на выходе выдают одни и те же результаты. Универсальным является такой язык программирования, на котором для любой *вычислимой функции* может быть составлена программа. Функция называется вычислимой, если для нее может быть состав-

лена программа на каком-либо из языков программирования. Такая формулировка проблемы несколько напоминает порочный круг, так как мы сталкиваемся с фундаментальным вопросом определения того, что такое *вычислимая функция*. Интуитивно понятно, что функцию можно назвать вычислимой, если существует какая-либо процедура, которая шаг за шагом может вычислить значение этой функции. При этом подразумевается, что процедура должна быть конечна во времени. Тем не менее, чтобы определить класс всех вычислимых функций, нам нужно предложить некий универсальный виртуальный компьютер или универсальный язык программирования, на котором эти функции могут быть выражены. Теперь проблема заключается в том, что мы не знаем, как определить, что язык универсален.

Оказывается, этот вопрос рассматривался еще до того, как появились первые компьютеры. В 30-е гг. математики исследовали задачу определения класса вычислимых функций. Некоторые ученые для решения этой задачи предложили использовать простые абстрактные машины, или автоматы, которые могли бы служить отправной точкой для определения класса вычислимых функций. Наиболее известной из них является *машина Тьюринга*, названная так в честь своего изобретателя, Алана Тьюринга (Alan Turing) [114].

В машине Тьюринга имеется только одна простая структура данных — линейно-упорядоченный массив переменной длины, называемый *рабочей лентой*. Каждый элемент, или *ячейка*, рабочей ленты содержит только один символ. Имеется также одна простая переменная-указатель, называемая *управляющей головкой*, которая в любой момент времени указывает на некоторую ячейку рабочей ленты. Машина Тьюринга управляется программой, в которую может входить только несколько простых операций.

1. Можно прочитать символ, хранящийся в ячейке рабочей ленты, на которую указывает управляющая головка, или записать в эту ячейку новый символ (то есть заменить старый символ новым). В зависимости от прочитанного символа программа может совершить условный переход. Можно использовать для формирования циклов и безусловные переходы goto. То есть по внутренней логике машина Тьюринга аналогична конечному автомату, который был рассмотрен ранее.
2. Положение управляющей головки может быть изменено таким образом, чтобы она указывала на ячейку рабочей ленты, расположенную слева или справа от текущей. В каждый момент времени рабочая лента состоит из конечного числа заполненных ячеек, но при сдвиге управляющей головки на позицию вне этих ячеек к ним автоматически справа или слева добавляется новая ячейка с записанным в нее пустым символом.

В начальный момент времени в ячейках рабочей ленты машины Тьюринга содержатся входные данные, а управляющая головка позиционирована на самую левую ячейку. Машина Тьюринга выполняет последовательность описанных выше простых операций, модифицируя содержимое ячеек рабочей ленты (при необходимости добавляя новые). Если в итоге она останавливается, то в ячейках рабочей ленты содержатся вычисленные результаты.

Машина Тьюринга — это чрезвычайно простая абстрактная машина. Обратите внимание на то, что она даже не способна производить арифметические действия.

Если вы захотите с ее помощью сложить два числа, то вам придется для этого создать программу, используя только указанные выше элементарные операции. Не допускаются никакие другие переменные или структуры данных. Для хранения данных используется одна только рабочая лента (но при этом важно учесть, что ее возможности хранения данных не ограничены).

Может ли машина Тьюринга делать *что-нибудь* полезное? Можно привести несколько примеров программ, которые убедят вас, что машину Тьюринга можно использовать по крайней мере для выполнения таких простых действий, как сложение и вычитание. Но на самом деле можно доказать гораздо более сильное утверждение: машина Тьюринга может производить *любые полезные действия* в области вычислений! Таким образом, мы попытаемся доказать, что любое вычисление может быть выражено в виде программы для машины Тьюринга и, следовательно, язык машины Тьюринга является универсальным языком, хотя в нем предусмотрена только одна структура данных (вектор) и не определены никакие арифметические операции, отсутствуют определения подпрограмм и прочие структуры, стандартные для обычных языков программирования.

Формальное выражение этой идеи известно как тезис Черча (Church) (того же Черча, которого мы упомянем в разделе 4.2). *Любая вычислимая функция может быть вычислена при помощи машины Тьюринга.* Этот тезис не является теоремой, которую можно доказать. Тезис Черча — это некоторая гипотеза, которую, в принципе, можно опровергнуть. Но для этого потребуется предъявить какую-нибудь функцию, которая может быть вычислена посредством другого языка программирования и не может быть вычислена на машине Тьюринга. Тем не менее исследованием тезиса Черча в течение многих лет занимались многие математики; были предложены многочисленные абстрактные и реальные вычислительные машины и языки программирования. Но каждый раз удавалось доказать, что очередной метод, предложенный для создания универсального языка или универсальной вычислительной машины, на самом деле является не более мощным, чем машина Тьюринга. То есть любая функция, которую можно было вычислить при помощи нового языка или новой машины, могла быть вычислена и с помощью машины Тьюринга.

При рассмотрении языков программирования следует иметь в виду, что машина Тьюринга эквивалента грамматике типа 0, которую мы обсуждали в предыдущем разделе этой главы. Достаточно легко показать, что любое состояние машины Тьюринга можно смоделировать при помощи некоторого вывода грамматики типа 0. Аналогично любая грамматика типа 0 распознается при помощи (недетерминированной) машины Тьюринга. Тот факт, что (в отличие от контекстно-свободных грамматик) недетерминированные и детерминированные машины Тьюринга эквиваленты, может вызвать некоторое удивление. То есть любое вычисление, проводимое при помощи недетерминированной машины Тьюринга, можно произвести, используя эквивалентную детерминированную машину Тьюринга.

Машина Тьюринга является примером того, что *для достижения универсальности не требуется почти никаких аппаратных средств, за исключением неограниченной памяти.* Даже чрезвычайно простой набор структур данных и операций оказывается достаточным для выражения любой вычислимой функции.

Проблема останова

Изучение машин Тьюринга и языков привело к некоторым другим важным результатам, в том числе к тому, что некоторые задачи являются *неразрешимыми*. Это означает, что для их решения не существует никакого общего алгоритма, даже в контексте описанных простых абстрактных машин.

Рассмотрим подробнее уже упоминавшийся ранее в этой главе вопрос о программе, определяющей, завершится ли когда-нибудь выполнение другой программы, написанной на языке C. Мы знаем, конечно, некоторые примеры программ на C, выполнение которых обязательно завершится. Например, программа

```
main()
{int i;
i=0;
}
```

безусловно будет завершена. Фактически завершается выполнение любой программы на C, которая состоит только из объявлений и операторов присваивания. Для того чтобы тестирующая программа могла определить такой тип программ, необходимо сделать следующее:

1) взять какой-либо компилятор C;
2) исключить из НФБ-грамматики, определяющей язык C, все правила, которые включают какое-либо ветвление, вызов процедур или операторы цикла;
3) запустить компилятор, используя только получившийся урезанный набор правил грамматики языка C.

Любая программа, откомпилированная при помощи такого модифицированного компилятора, может состоять только из операторов присваивания и неизбежно завершится после выполнения. Интерес представляют те программы, которые не компилируются при помощи такого компилятора. Некоторые, очевидно, являются корректными, их выполнение будет завершено (например, те программы, в которых содержатся корректно работающие циклы); но программы, в которых циклы работают неправильно, могут зациклиться и выполняться бесконечно. Проблема заключается в том, что не существует каких-либо общих критериев, позволяющих определить различия между этими типами программ.

Предположим, что у вас имеется такая универсальная тестирующая программа и вы хотите с ее помощью проверить некоторую очень длинную программу (обозначим ее A). Допустим, вы задали предельно допустимое время проверки как 3 года; если в течение этого времени тестирующая программа не сможет прийти к выводу, что программа A завершится, то будет считаться, что данная программа A никогда не завершится.

По прошествии трех лет вы останавливаете работу тестирующей программы и делаете вывод, что программа A никогда не завершится. Но, возможно, если бы вы подождали еще 10 минут, ответ был бы найден. Суть проблемы в том, что если речь идет о вещах, не принадлежащих вычислимому множеству, мы просто не знаем, когда следует остановить выполнение программы и получить ответ.

Наши интуитивные рассуждения связаны с так называемой *проблемой останова*: существует ли какой-либо общий алгоритм, позволяющий определить, остано-

вится ли через какое-нибудь время машина Тьюринга, если в качестве входных данных ей была передана какая-либо конкретная строка символов? Исследования самого Тьюринга, проведенные в 1936 г., показали, что проблема останова неразрешима: не существует алгоритма решения задачи останова, общего для всех машин Тьюринга и для всех входных строк.

Для подтверждения неразрешимости какой-либо задачи мы показываем, что она эквивалентна задаче останова. Действительно, если бы рассматриваемая нами задача была разрешима, то была бы разрешима и задача останова. Но мы знаем, что это не так. Следовательно, исходная задача также неразрешима.

Исследование этих простых универсальных языков и машин приводит нас к выводу, что любой язык программирования, который можно было бы использовать на практике, является универсальным, если проигнорировать ограничения, связанные со временем выполнения и объемом требуемой памяти. Например, если какой-нибудь программист непоколебим в своем решении использовать только один язык программирования, например LISP, и утверждает, что «все можно сделать на LISP», то он, строго говоря, прав. Можно сделать все, хотя какие-то вещи, возможно, будет сделать нелегко. Различия между языками программирования не являются *качественными* — то есть они заключаются не в том, что решение каких-то задач возможно только при использовании определенных языков; эти различия носят *количественный* характер и сводятся к тому, насколько легко, красиво и эффективно могут быть решены задачи на различных языках.

Неоднозначность

Иерархия типов грамматик Хомского является примером обычного для теоретических исследований явления: для описания какой-либо встречающейся на практике ситуации можно построить множество теоретических моделей, описывающих ее различными способами. Некоторые из этих моделей оказываются менее полезными, чем другие, и легко забываются после непродолжительного их изучения. Другие модели, в которых отражены какие-то важные для практики стороны изучаемой проблемы, становятся общепринятыми. Контекстно-зависимые грамматики и грамматики с фразовой структурой в иерархии Хомского оказались неудачными моделями языков программирования. Хотя языки, определяемые грамматиками этих типов, являются более мощными, в то же время они сложнее для понимания, анализа и практического использования. Впоследствии (при разработке новых грамматик) были сделаны попытки преодолеть ограничения, характерные для этих моделей.

У НФБ-грамматик имеется ряд достоинств. Их довольно просто использовать на практике, и они являются достаточно мощными, чтобы выразить большинство (но не все) синтаксических конструкций, которые необходимы в языках программирования (лингвисты считают, что для естественных языков такие грамматики менее полезны). К преимуществам НФБ-грамматик относится и то, что с помощью математического анализа такой грамматики легко обнаружить неявные свойства определяемого ею языка.

Например, имеет большое практическое значение вопрос: является ли данная НФБ-грамматика языка программирования неоднозначной (то есть допускает ли

эта грамматика возможность нескольких вариантов синтаксического разбора программ или существует только один такой вариант для каждой программы)? Обычно каждый вариант синтаксического разбора соответствует определенному семантическому варианту прочтения программы, поэтому различные варианты синтаксического разбора приводят к различным значениям одной и той же программы. Возникает вопрос: можно ли найти какую-нибудь общую для всех грамматик процедуру, определяющую, является ли данная грамматика неоднозначной? Теоретические исследования привели к неожиданному результату: искать такую процедуру бессмысленно, ее просто нет. Формальное утверждение звучит следующим образом: вопрос об однозначности или неоднозначности НФБ-грамматики *неразрешим*; не существует никакой общей процедуры, позволяющей ответить на этот вопрос для любой НФБ-грамматики. Этот результат не может не разочаровывать, так как в случае сложной НФБ-грамматики, в которой имеются сотни различных правил, было бы очень удобно иметь такую программу, способную проверить эту грамматику на однозначность. В большинстве случаев мы можем написать программу, которая определяет, однозначна ли *данная* грамматика; неразрешимость означает, что невозможно написать одну тестирующую программу, которая давала бы ответ для *всех* программ. Для некоторых грамматик такой ответ не будет получен, как бы долго ни работала тестирующая программа.

4.1.3. Сложность алгоритма

Иерархия Хомского до сих пор остается интригующей темой для исследований в теории программирования. Хотя лексический анализ и грамматический разбор хорошо изучены, все же осталось еще множество вопросов, требующих решения. В этом разделе представлен обзор таких нерешенных вопросов.

Таблица 4.1. Классы грамматик и абстрактных машин

Уровень в иерархии Хомского	Класс грамматики	Класс машин
0	Неограниченные	Машина Тьюринга
1	Контекстно-зависимые	Линейно-ограниченный автомат
2	Контекстно-свободные	Автомат с магазинной памятью
3	Регулярные	Конечный автомат

Грамматика и машины. Мы говорили, что существует определенное соответствие между классом грамматик и классом абстрактных машин. В табл. 4.1 вы найдете краткую сводку по этой теме, а на рис. 4.2 соответствие классов машин и грамматик представлено схематически.

1. Конечный автомат (рис. 4.2, *а*) состоит из конечного графа состояний и рабочей ленты с возможностью перемещения по ней только в одном направлении. При совершении очередной операции автомат считывает следующий символ с ленты и входит в новое состояние (узел на графе).
2. В автомате с магазинной памятью (рис. 4.2, *б*) в отличие от конечного автомата добавляется стек. При каждой операции автомат считывает следую-

щий символ с рабочей ленты и верхний символ стека, записывает новый символ в стек и входит в новое состояние.

3. Линейно-ограниченный автомат (рис. 4.2, *в*) похож на конечный автомат, но он может также записывать символы в ячейки рабочей ленты и перемещаться по ней в обоих направлениях.
4. Машина Тьюринга (рис. 4.2, *г*) похожа на линейно-ограниченный автомат, но только в данном случае лента является неограниченной в обоих направлениях.

Все классы машин, кроме линейно-ограниченного автомата, уже обсуждались ранее. *Линейно-ограниченный автомат, или ЛО-автомат* (Linear-Bounded Automation, LBA), определяется как машина Тьюринга, в которой можно использовать только ту часть рабочей ленты, которая содержит входные данные. Таким образом, объем памяти возрастает с увеличением объема входных данных, и такой автомат способен распознавать более сложные цепочки. Тем не менее по сравнению с машиной Тьюринга возможности этого автомата в отношении хранения информации не так велики, так как в данном случае рабочая лента не увеличивается неограниченно.

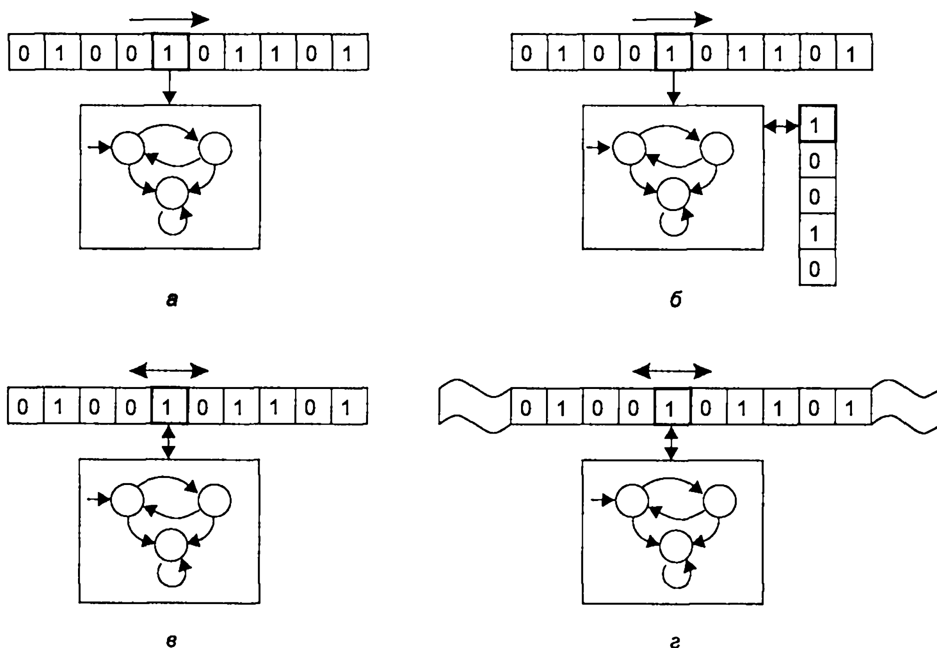


Рис. 4.2. Модели абстрактных машин: *а* — конечный автомат; *б* — автомат с магазинной памятью; *в* — линейно-ограниченный автомат; *г* — машина Тьюринга

При обсуждении вычислительных возможностей детерминированной и недетерминированной версий абстрактной машины каждого типа мы получили следующие интересные результаты.

Тип машины	Сравнение версий
Конечный автомат	Одинаковые
Автомат с магазинной памятью	Неодинаковые
Линейно-ограниченный автомат	?
Машина Тьюринга	Одинаковые

Как видно из этой таблицы, для языков типа 3 и 0 отсутствие детерминизма ничего не добавляет к вычислительным возможностям соответствующей абстрактной машины. Функциональные возможности недетерминированного автомата совпадают с возможностями эквивалентного детерминированного автомата. Однако мы знаем, что для языков типа 2 недетерминированная версия абстрактной машины обладает большими возможностями. Мы можем распознавать строки-палиндромы при помощи недетерминированного автомата с магазинной памятью (см. раздел 3.3.5), так как этот автомат способен определять середину строки. Но при помощи недетерминированного автомата с магазинной памятью эта задача не решается. Детерминированный автомат с магазинной памятью распознает детерминированные контекстно-свободные языки, которые являются не чем иным, как языками LR(k), образующими основу для теории синтаксического разбора, используемой при построении компиляторов.

К сожалению, для линейно-ограниченного автомата неизвестно, как соотносятся в смысле своих функциональных возможностей детерминированная и недетерминированная версии ЛО-автомата. Этой задачей ученые занимаются вот уже более тридцати лет.

Полиномиальное время вычислений. Сложность решения проблемы детерминизма для ЛО-автоматов привела к возникновению специального раздела теоретической информатики, посвященного исследованию сложности алгоритмов. Детерминизм языка программирования можно соотнести с *полиномиальным временем* вычисления. Этот термин означает следующее: если длина введенной строки равна n, то абстрактная машина выполнит обработку этой строки за время, которое можно оценить сверху как $p(n)$, где p — это некоторый полином (то есть $p(x) = a_1x^n + a_2x^{n-1} + \ldots + a_{n-1}x + a_n$). В противоположность этому можно показать, что в недетерминированном случае вычисление невозможно осуществить за полиномиальное время. Тогда задача об эквивалентности детерминированной и недетерминированной версий ЛО-автомата сводится к следующей задаче: если некоторое вычисление было выполнено за неполиномиальное время (НП), существует ли такая машина, на которой это же вычисление может быть выполнено за полиномиальное время (П)? Иначе говоря, верно ли, что НП = П? Эта задача для ЛО-автомата называется задачей *НП-полноты*, а любая другая задача, эквивалентная этой, называется *НП-полной*. Следовательно, задача об эквивалентности детерминированной и недетерминированной версий ЛО-автоматов является НП-полной.

Существует множество других задач, связанных с уже упомянутыми, для которых можно доказать, что они являются НП-полными; но пока у нас нет ответа на основной вопрос. Многие теоретики полагают, что ответа вообще не существует. Он просто находится за пределами аксиоматики используемого нами математического формализма.

Значение формальных моделей

Тем, кто занимается практическим программированием, часто не хватает понимания важности теоретических моделей и исследований в области разработки языков и технологий программирования. Может сложиться впечатление, что та степень абстрагирования от реальной задачи, которая необходима на первой стадии построения теоретической модели, иногда приводит к такому упрощению этой задачи, что теряется весь ее смысл. Надо сказать, что довольно часто так и происходит. Выбрав неверную теоретическую модель, можно ее исследовать и получить какие-то результаты, которые, однако, невозможно будет преобразовать в решение исходной практической задачи. Но если, как мы постарались показать в этом разделе, была найдена правильная модель, то ее теоретическое исследование может дать результаты, имеющие серьезное практическое значение.

4.2. Семантика языка

О синтаксисе языка программирования известно довольно много; менее изучен вопрос корректного определения семантики языка. Руководство по использованию языка программирования должно включать описание каждой конструкции языка как по отдельности, так и в совокупности с другими конструкциями. Задача корректного определения семантики похожа на задачу определения синтаксиса. В языке имеется множество различных конструкций, точное определение которых необходимо как программисту, использующему язык, так и разработчику реализации этого языка. Программисту эти сведения нужны для того, чтобы писать правильные программы и заранее знать результат выполнения любых операторов программы. Разработчику корректные определения конструкций необходимы для создания правильной реализации языка.

В большинстве руководств определение семантики дается в виде обычного текста. Как правило, сначала при помощи какой-либо формальной грамматики (например, НФБ-грамматики) дается определение синтаксиса конструкции, а затем для пояснения семантики приводятся несколько примеров и небольшой пояснительный текст. К сожалению, смысл этого текста часто неоднозначен, так что разные читатели могут понимать его по-разному. Программист может получить ошибочное представление о том, что именно будет делать написанная им программа при выполнении, а разработчик может реализовать какую-либо языковую конструкцию иначе, чем разработчики других реализаций того же языка. Как и в случае синтаксиса, нужен какой-то метод, позволяющий дать удобочитаемое, точное и лаконичное определение семантики всего языка.

Задача определения семантики языка программирования рассматривается теоретиками так же долго, как и задача определения синтаксиса, но в данном случае гораздо труднее найти удовлетворительное решение. Было разработано множество различных методов формального определения семантики. Ниже мы приводим описания некоторых из них.

Грамматические модели. Некоторые ранние попытки добавить корректное определение семантики к языку программирования делались путем добавления расширений к НФБ-грамматике, определяющей этот язык. Дополнительную ин-

формацию о семантике можно было извлечь из дерева синтаксического разбора. Мы коротко обсудим атрибутивные грамматики как способ получения этой дополнительной информации.

Императивные (операционные) модели. *Операционное* определение языка программирования дает описание того, как составленные на данном языке программы выполняются на виртуальном компьютере. Обычно виртуальный компьютер определяется как автомат, но гораздо более сложный по сравнению с обычными моделями автоматов, которые используются при изучении синтаксиса и осуществлении синтаксического разбора. Внутренние состояния этого автомата соответствуют состояниям программы при ее выполнении; это означает, что в состояние автомата входят значения всех переменных, выполняемая программа и различные вспомогательные системные структуры данных. Для определения возможных изменений внутреннего состояния автомата в результате выполнения одного оператора программы используется набор формально определенных операций. Вторая часть определения задает способ трансляции текста программы в исходное состояние автомата. Начиная с этого исходного состояния, автомат в соответствии с определяющими его правилами последовательно переходит к следующим состояниям, пока не достигнет конечного. Такое операционное определение языка программирования может представлять собой достаточно прямую абстракцию возможной фактической реализации языка. С другой стороны, это определение может представлять и более абстрактную модель, которую можно использовать как основу для программно моделируемого интерпретатора языка, но не для фактической реализации.

В 70-е гг. была разработана операционная модель языка под названием *Vienna Definition Language* (VDL) — метаязык, предназначенный для описания других языков. В этой модели дерево синтаксического анализа включает в себя также машинный интерпретатор. Состояние вычисления входит в дерево программы, а также в дерево, описывающее все данные для конкретной машины. Очередной оператор переводит дерево в новое состояние.

Аппликативные модели. Аппликативное определение языка пытается непосредственно сконструировать определение функции, которую вычисляет каждая программа, написанная на этом языке. Такое определение языка строится как иерархия определений функций, которые вычисляются каждой отдельной программной конструкцией. По аналогии с аппликативными языками, которые обсуждались в первой главе, этот метод определения языка представляет собой аппликативный подход к моделированию семантики.

В программе любая элементарная операция или операция, определенная программистом, представляет собой некоторую математическую функцию. Структуры управления последовательностью действий могут быть использованы для композиции этих функций в более крупные последовательности, представленные в тексте программы выражениями и операторами. Линейные последовательности операторов и условное ветвление легко могут быть представлены функциями, составленными из функций, которые соответствуют отдельным компонентам этих конструкций. Цикл обычно представляется посредством рекурсивной функции, составленной из компонентов, входящих в тело цикла. В конце концов образуется функциональная модель всей программы. Примерами такого подхода к определе-

нию семантики являются метод *денотационной семантики* Скотта (Scott) и Стрэчи (Strachey) и метод *функциональной семантики* Миллза (Mills). В разделе 4.2.2 вы найдете краткое введение в денотационную семантику.

Аксиоматические модели. Данный метод распространяет на программы область применения исчисления предикатов. Семантику каждой синтаксической конструкции языка можно определить как некий набор аксиом или правил вывода, который можно использовать для вывода результатов выполнения этой конструкции. Чтобы понять смысл всей программы (то есть разобраться, что и как она делает), эти аксиомы и правила вывода следует использовать так же, как при доказательстве обычных математических теорем. В предположении, что значения входных переменных удовлетворяют некоторым ограничениям, аксиомы и правила вывода могут быть использованы для получения (вывода) ограничений на значения других переменных после выполнения каждого оператора программы. В конце концов, когда программа выполнена, мы получаем доказательство того, что вычисленные результаты удовлетворяют необходимым ограничениям на их значения относительно входных значений. То есть доказано, что выходные данные представляют значения соответствующей функции, вычисленной по значениям входных данных. Примером описанного подхода является метод *аксиоматической семантики*, разработанный Хоором (Hoare) (см. раздел 4.2.4).

Модели спецификаций. В модели спецификаций мы описываем отношение между различными функциями, реализующими программу. Пока нам удается показать, что реализация подчиняется этому отношению между любыми двумя функциями, мы можем утверждать, что она корректна по отношению к спецификации.

Алгебраический тип данных является одним из видов формальной спецификации. Например, если вы пишете программу, реализующую стеки, то действия push (записать в стек) и pop (прочитать из стека) имеют противоположное действие в том смысле, что если для некоторого заданного стека S выполнить действие push, а затем немедленно — действие pop, то в итоге получится исходный стек. Это можно сформулировать в виде аксиомы:

```
pop(push(S.x)) = S
```

Любая реализация, которая сохраняет это свойство (а также некоторые другие), является корректной реализацией стека. В разделе 4.2.5 мы представим краткий обзор алгебраических типов данных.

Формальное определение семантики становится общепринятой частью определения нового языка. Стандартное описание языка PL/I включает в себя VDL-подобную нотацию, описывающую семантику операторов PL/I, а для языка Ada было разработано определение на основе денотационной семантики. Тем не менее изучение формальных определений семантики не оказало такого сильного влияния на практическое определение языков, как изучение формальных грамматик — на определение синтаксиса. Ни один из методов определения семантики не оказался полезным ни для пользователя, ни для разработчика языка. Операционные модели достаточно удобны для создания формальной модели реализации и могут быть полезны разработчику, но для пользователя эти модели не имеют большого значения, так как в них слишком много ненужных ему подробностей. Разработчик вряд ли сможет руководствоваться функциональными и денотационными моделями, а для пользователя они, как правило, оказываются слишком сложными, чтобы их можно было

использовать непосредственно. Пользователю легче понять аксиоматические модели, но при попытке составить полное определение языка обычно они становятся чрезвычайно сложными, а для разработчика эти модели и вовсе непригодны.

В следующем разделе вы найдете краткое описание атрибутивной грамматики как одной из форм семантической модели языка программирования. В разделе 4.2 мы описываем другие семантические модели.

4.2.1. Атрибутивные грамматики

Одной из первых попыток разработки семантической модели языка программирования была концепция *атрибутивной грамматики*, предложенная Дональдом Кнутом [64]. Идея заключалась в том, чтобы сопоставить каждому узлу дерева синтаксического разбора данной программы некоторую функцию, задающую семантическое содержание данного узла. Атрибутивные грамматики создавались путем добавления функций (*атрибутов*) к каждому правилу грамматики.

Унаследованный атрибут — это функция, которая сопоставляет нетерминальные значения в данном узле дерева с нетерминальными значениями, расположенными на более высоком уровне дерева. Иными словами, функциональное значение для нетерминальных символов, расположенных в правой части любого правила, является функцией нетерминальных символов, расположенных в его левой части.

Синтезированный атрибут — это функция, которая соотносит нетерминальные символы, расположенные в левой части правила, со значениями нетерминальных символов, расположенными в правой части правила. Такие атрибуты передают информацию вверх по дереву (то есть они синтезированы на основе информации, взятой из нижних уровней дерева).

Рассмотрим простую грамматику для арифметических выражений:

```
E → T | E + T
T → P | T × P
P → I | (E)
```

Семантику этого языка можно определить посредством набора отношений между нетерминальными символами грамматики. Например, значение любого выражения, определяемого этой грамматикой, выводится при помощи следующих функций.

Правило	Атрибут
E → E + T	значение (E_1) = значение (E_2) + значение (T)
E → T	значение (E) = значение (T)
T → T × P	значение (T_1) = значение (T_2) × значение (P)
T → P	значение (T) = значение (P)
P → I	значение (P) = значение числа I
P → (E)	значение (P) = значение (E)

Здесь нижние индексы 1 и 2 обозначают соответственно ссылку на первый и второй одинаковые нетерминальные символы в данном правиле. На рис. 4.3 представлен пример дерева с атрибутами, которое дает значение выражения 2 + 4 × (1 + 2).

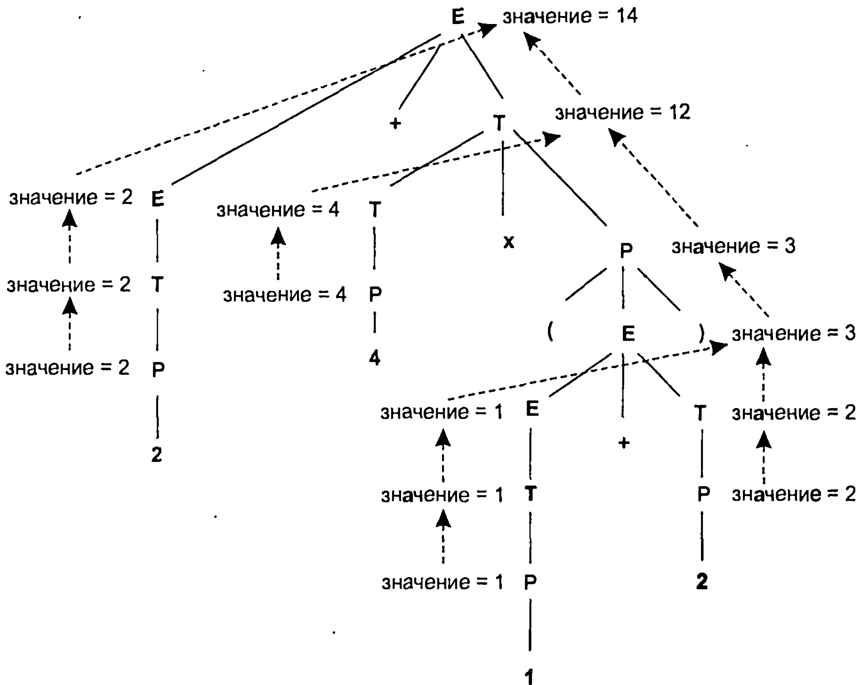


Рис. 4.3. Пример дерева со значениями атрибутов

Атрибутивные грамматики можно использовать для передачи семантической информации по синтаксическому дереву. Например, можно с помощью правил объявлений собрать всю информацию об объявлениях и информацию из получившейся таблицы символов передавать вниз по дереву для использования при генерации кода для выражений.

Например, для создания множества имен, объявленных в программе, можно добавить следующие атрибуты к нетерминальным символам <decl> и <declaration>.

Правило	Атрибут
<declaration> ::= <decl><declaration>	$decl_set(declaration_1) = decl_name(decl) \cup decl_set(declaration_2)$
<declaration> ::= <decl>	decl_set(declaration) = decl_name(decl)
<decl> ::= **declare** x	decl_name(decl) = x
<decl> ::= **declare** y	decl_name(decl) = y
<decl> ::= **declare** z	decl_name(decl) = z

Синтезированный атрибут decl_set, связанный с нетерминальным символом <declaration>, всегда содержит множество имен, объявленных в данной программе. Этот атрибут можно передавать вниз по дереву через унаследованный атрибут и использовать при генерации кода для введенных данных.

Если в грамматике имеются только синтезированные атрибуты (как в приведенном выше примере), то компилятор может вычислить их в момент генерации

дерева синтаксического разбора на соответствующем этапе трансляции. Так работают системы, подобные YACC. Каждый раз, когда YACC определяет, какое применить правило (продукцию) НФБ-грамматики, выполняется подпрограмма (например, атрибутивная функция), которая дополняет дерево синтаксического разбора семантической информацией.

4.2.2. Денотационная семантика

Денотационная семантика — это формальная аппликативная модель для описания семантики языков программирования. Прежде чем обсуждать денотационную семантику, введем понятие λ-исчисления, которое представляет собой более простую функциональную модель, разработанную еще в 30-е гг. как способ объяснения математических вычислений. На основе λ-исчисления могут быть построены более сложные структуры, включая концепцию типов данных и семантику языков программирования. Далее в этой главе (в разделе 4.2.3) мы опишем язык ML как пример функционального языка, основанного на некоторой функциональной разновидности денотационной семантики.

λ-исчисление

Возможно, самой первой моделью семантики языка программирования было *λ-исчисление*, предложенное в 30-х гг. А. Черчем в качестве теоретической модели вычислений, сопоставимой с машиной Тьюринга (см. раздел 4.1.3). Оказалось, что λ-исчисление хорошо моделирует вызов функций в языках программирования, хотя его появление опередило самые первые компьютеры на несколько лет, а языки программирования — на пятнадцать. Фактически семантика обращения к функциям в языках ALGOL и LISP унаследована от λ-исчисления; механизм подстановки в λ-исчислении является прямым отображением определенного в языке ALGOL механизма передачи параметра по имени (см. раздел 9.3). Скотт [98] расширил эту модель, превратив ее в общую теорию типов данных, которая сегодня известна под названием *денотационной семантики*. Система обозначений, принятая в денотационной семантике, оказала влияние на разработку теории типов данных в языке ML. Полное описание λ-исчисления и денотационной семантики не входит в задачу авторов этой книги; тем не менее мы хотели бы представить краткий обзор этих концепций и показать их связь с разработкой языков программирования.

λ-выражения определяются рекурсивно при помощи следующих правил:

1. Если x является именем переменной, то x является λ-выражением.
2. Если M является λ-выражением, то λx.M является λ-выражением.
3. Если F и A являются λ-выражениями, то (FA) — также λ-выражение. Здесь F — это *оператор*, а A — *операнд*.

λ-выражения можно определить и при помощи контекстно-свободной грамматики:

```
λ_выражение → идентификатор | λидентификатор.λ_выражение | (λ_выражение λ_выражение)
```

Далее приведены примеры λ-выражений, сгенерированных этой грамматикой:

```
x           λx.x                λx.y
λx.(xy)     (λx.(xx)λx.(xx))    λx.λy.x
```

Переменные могут быть связанными или свободными. Интуитивно понятно, что *связанная* переменная — это локально объявленная переменная, а свободная переменная не имеет никакого объявления. Переменная x в λ-выражении x является *свободной* переменной. Если x в M является свободной, то в λx.M она связана. Переменная x является свободной в (F A), если она свободна в F или в A.

Имя любой связанной переменной можно поменять так же, как имена параметров функций. Так, λ-выражение λx.x эквивалентно λy.y, λ-выражение λx.λx.x эквивалентно λx.λy.y, поскольку переменная x связана с самым правым λx. Неформально можно сказать, что связанные переменные являются параметрами функции, описываемой λ-выражением; свободные же переменные являются глобальными. Эта аналогия показывает, что λ-выражение просто аппроксимирует концепцию процедур и подпрограмм для большинства алгоритмических языков типа Pascal, C, FORTRAN или Ada.

Операции над λ-выражениями

Для λ-выражений определена только одна операция — операция *редукции*. Если (F A) — λ-выражение и F = λx.M, то во всех случаях, где в M встречается свободная переменная x, вместо нее можно подставить A. Записывается это следующим образом: (λx.M A) ⇒ M'. Эта операция аналогична подстановке фактического параметра вместо формального параметра в вызове функции.

Ниже приведены некоторые примеры редукции в λ-выражениях:

```
(λx.x y)              ⇒ y
(λx.(xy) y)           ⇒ (yy)
(λx.(xy) λx.x)        ⇒ (λx.x y)              ⇒ y
(λx.(xx) λx.(xx))     ⇒ (λx.(xx) λx.(xx))     ⇒ ...
```

Заметим, что операция редукции не обязательно приводит к упрощению λ-выражения; четвертый пример иллюстрирует ситуацию, в которой редукция не завершается — иными словами, выражение не редуцируется. Исследование этих примеров приводит нас к формулировке *свойства Черча–Россера* (Rosser): если две различные редукции λ-выражения завершаются, то они являются членами одного класса значений. Иначе говоря, если для λ-выражения M имеются две редукции M ⇒ P и M ⇒ Q, то существует единственное λ-выражение R, такое что P ⇒ R и Q ⇒ R. Выражение R называется ***нормальной формой*** для класса значений, представителем которого является M.

Передача параметров в λ-выражениях. Существует два стандартных подхода к редукции любого λ-выражения:

1) сначала редуцировать самый внутренний левый член;
2) сначала редуцировать самый внешний левый член.

Например, в λ-выражении (λy.(yy) (λx.(xx) a)) самый внешний член — это все выражение, а самый внутренний — (λx.(xx) a). Здесь возможны две последовательности редукций.

- Сначала редуцируется самый внешний член:
  (λy.(yy)(λx.(xx)a)) ⇒ ((λx.(xx) a)(λx.(xx) a)) ⇒ ((aa)(aa))
- Сначала редуцируется самый внутренний член:
  (λy.(yy)(λx.(xx)a)) ⇒ ((λy.(yy)(aa)) ⇒ ((aa)(aa))

Хотя ответы совпадают, редукция происходит по-разному: в первом случае функция (λx.(xx) a) подставляется вместо параметра y и для каждого экземпляра y вычисляется значение этой функции. Такая редукция аналогична механизму вызова параметра по имени, описанному в разделе 9.3. Во втором случае сначала вычисляется значение константы (aa), которое затем подставляется вместо переменной y. Такая редукция представляет механизм вызова параметра по значению.

Если имеется нормальная форма, то редукция, произведенная по первому способу (то есть аналогичная вызову по имени), сведется к этой форме; хотя, как было показано ранее, ни один из способов редукции не гарантирует ее завершение. Вызов по имени — форма отложенного вычисления (см. раздел 8.2.2), поэтому требуется вычислить только выражения, используемые в окончательном решении. При вызове по значению, напротив, вычисляются значения всех аргументов, даже если они в итоге не будут использованы (например, в случае, если выражение не редуцируется).

Приведенные выше редукции можно использовать для создания простых примеров выражений, которые редуцируются при использовании первого способа (вызов по имени) и не редуцируются при использовании второго способа (вызов по значению). Единственное, что для этого нужно, — сконструировать нередуцируемое λ-выражение как аргумент, ссылки на который отсутствуют. Мы уже знаем, что выражение (λx.(xx) λx.(xx)) не редуцируется, а на параметр y в выражении λy.z, очевидно, отсутствуют ссылки. Следовательно, для выражения

```
(λy.z (λx.(xx)λx.(xx)))
```

при использовании редукции первого типа (вызов по имени) легко получить нормальную форму, которой является z; но если вы примените вызов по значению, то это выражение окажется нередуцируемым.

Моделирование математики при помощи λ-выражений

λ-исчисление изначально было создано как логическая модель вычислений. Можно использовать λ-выражения для моделирования арифметики как мы ее понимаем. Сначала при помощи λ-выражений моделируется исчисление предикатов; затем на основе построенного исчисления предикатов можно моделировать целые числа.

Булевы значения. Булевы значения моделируются как λ-выражения следующим образом.

Истина (true, T) определяется как: λx.λy.x. (Понять это выражение станет легче, если мы установим, что означает Истина: из пары значений следует выбрать первое. Это показано на примере, который мы рассмотрим позже.)

Ложь (false, F) определяется как: λx.λy.y. (Из пары значений следует выбрать второе.)

Используя определения этих объектов, мы получаем, что следующие свойства истинны:

```
((T P)Q)⇒ P, то есть ((T P)Q) ⇒ ((λx.λy.x P)Q) ⇒ (λy.P Q) ⇒ P;
((F P)Q)⇒ Q, то есть ((F P)Q) ⇒ ((λx.λy.y P)Q) ⇒ (λy.y Q) ⇒ Q.
```

Имея определения для констант T и F, можно определить следующие логические функции:

```
not (логическое отрицание)     = λx.((xF)T);
and (конъюнкция)               = λx.λy.((xy)F);
or (дизъюнкция)                = λx.λy.((xT)y).
```

Теперь нам предстоит показать, что наша интерпретация булевых значений согласуется с правилами логики предикатов. Например, применение функции not к T (Истина) должно дать F (Ложь), и наоборот. Действительно,

```
(not T) = (λx.((x F)T)T) ⇒ ((T F)T) ⇒ F;
(not F) = (λx.((x F)T)F) ⇒ ((F F)T) ⇒ T.
```

Можно показать аналогичным образом, что определенные нами функции и и или также обладают всеми необходимыми свойствами данных логических операций.

Целые числа. Используя определенные нами булевы функции, можно перейти к определению целых чисел:

```
0 = λf.λc.c
1 = λf.λc.(fc)
2 = λf.λc.(f(fc))
3 = λf.λc.(f(f(fc)))
... ...
```

Здесь c соответствует нулевому элементу, а f — это функция, последовательно добавляющая к c единицу. С помощью этих определений можно определить обычные арифметические операции: целое число N записывается посредством λ-выражения как (N a), то есть λc.(a...(a c)...). Применяя редукцию к ((N a)b), мы получаем (a...(a b)...).

Рассмотрим λ-выражение ((M a)((N a)b)), применив константу ((N a)b) к λ-выражению (M a). Подстановка ((N a)b) вместо c в (M a) дает (a...(a b)...), причем теперь уже в этом списке присутствует (M+N) символов a. Таким образом, мы только что показали, что λ-выражения можно складывать:

```
[M+N] = λa.λb((M a)((N a)b))
```

Иначе говоря, мы определили операцию + как

```
+ = λM.λN.λa.λb.((M a)((N a)b))
```

Аналогичным образом можно показать, что

Умножение	$[M \times N] =$	λa.(M(N a))
Возведение в степень	$[M^N] =$	(N M)

Продолжая в том же духе, мы можем определить все вычислимые математические функции. Однако давайте применим эти идеи к семантике языков программирования.

Моделирование языков программирования

Используя полученные сведения о λ-выражениях, можно пойти дальше и применить λ-выражения к моделированию типов данных; после этого мы можем сделать следующий шаг и получить модель, включающую также и семантику языков программирования.

Когда к λ-выражению применяется операция редукции, оно сводится либо к некоторым константам, либо к другим λ-выражениям. Таким образом, все λ-выражения являются решениями следующего функционального уравнения:

λ-выражение = константа + (λ-выражение ⇒ λ-выражение).

Не углубляясь в детали денотационной семантики, мы просто констатируем, что *тип данных* — это решение такого уравнения. Заметим, однако, что между этой формальной моделью типа данных и определениями типов в языке ML имеется

прямое соответствие. А именно список целых чисел (конструктор Mylist) определяется в ML следующим образом:

```
datatype Mylist = var of int |
                 listitem of int * Mylist;
```

Здесь используются конструкторы var и listitem, по своему назначению аналогичные λ-выражениям.

С помощью этой концепции мы можем определить модель *денотационной семантики* простого языка программирования. Эта модель является разновидностью операционной семантики, поскольку мы прослеживаем выполнение каждого типа операторов, чтобы определить его воздействие на интерпретатор более высокого уровня. Однако в отличие от традиционных интерпретаторов, которые описаны в главе 2, мы рассматриваем программу как некоторую функцию аналогично ранней трактовке λ-исчисления. Каждый оператор языка — это функция, и мы моделируем выполнение последовательных операторов, определяя, какой вид должна иметь композиция функций для двух операторов.

Для каждого типа операторов (Stmt) можно написать уравнение определения типа данных:

```
Stmt =  (Id × Exp)              Область операторов присваивания
        +(Stmt × Stmt)          Область последовательностей
        +(Exp × Stmt × Stmt)    Область условных операторов
        +(Exp × Stmt)           Область итераций
```

Как в случае с любым интерпретатором, нам необходимо представлять себе его виртуальный компьютер. В данном случае мы предполагаем, что каждый идентификатор ссылается на определенную область памяти. В этом простом примере мы не используем замещение имен, передачу параметров и указатели. Следовательно, каждому идентификатору соответствует некая область памяти, в которой содержится значение этого идентификатора. Иначе говоря, мы можем моделировать нашу концепцию памяти функцией, которая для каждого идентификатора возвращает его значение. Такую функцию мы будем называть *памятью*.

Итак, память для программы — это функция, сопоставляющая каждой ее области (или идентификатору) соответствующее значение. Тогда результат выполнения оператора можно представить как некоторую модификацию памяти. Мы имеем новую функцию (аналогичную предыдущей), которая теперь сопоставляет каждому идентификатору некое значение, которое может совпадать или не совпадать с предыдущим. Таким образом, выполнение оператора мы рассматриваем как суперпозицию двух функций: функции, представляющей память, и функции, которая описывает выполнение данного оператора.

Мы рассматриваем память как отображение идентификаторов на область значений, хранящихся в памяти, и записываем это с помощью сигнатуры вида id → value, где id — это множество идентификаторов данного языка, а value — множество их значений. Мы называем это *состоянием программы*.

При определении семантики языка программирования нам следует представить три основные функции.

1. Необходимо описать семантику *программы*. В данном случае синтаксис программы определяет функцию, которая, вообще говоря, сопоставляет неко-

торому числу какое-то иное число. Другими словами, мы ищем функцию M с сигнатурой

```
M : prog → [num → num]
```

2. Необходимо описать оператор языка. Каждый оператор отображает данное состояние в другое, то есть

   ```
   C : stmt → [state → state]
   ```

   Эта запись означает, что каждый синтаксический оператор stmt представляет собой функцию, сопоставляющую одному состоянию state программы другое состояние state. Следовательно, каждый конкретный оператор берет данное состояние программы (то есть отображение из области идентификаторов на область их значений) и производит новое отображение идентификаторов на их значения, являющееся результатом действия всех предыдущих операторов, включая и тот, о котором идет речь.

3. Функция C, описывающая действие оператора, зависит от значений различных выражений, поэтому нам необходимо понимать процесс вычисления выражений в рассматриваемом языке. Любое выражение — это набор синтаксических единиц, в том числе идентификаторов, каждому из которых сопоставлено некое значение в соответствующей области памяти. Таким образом, для данного выражения сопоставление набора входящих в него идентификаторов и других синтаксических единиц с набором значений этих идентификаторов позволяет получить значение этого выражения. Тогда для вычисления выражений мы получаем следующую сигнатуру:

   ```
   E : exp → [state → eval]
   ```

В качестве простого примера рассмотрим вычисление выражения exp, представляющего собой сложение двух величин a+b. Тогда функция E для вычисления выражения такова:

```
E(a+b) : state → eval:
```

применение этой функции к состоянию s дает функцию

```
E(a + b)(s) = a(s) + b(s)
```

Обычно пишется не E(a+b), а E{{a+b}}, то есть синтаксические аргументы помещаются не в круглые скобки, а в фигурные.

Теперь мы можем моделировать области для нашего языка программирования:

```
state = id → value     состояния программы
id                     идентификаторы
value = eval           значения
eval = num + bool      значения выражений
num                    целые числа
bool                   булевы значения
exp                    выражения
stmt                   операторы
```

Для того чтобы определить язык, нужно определить функцию типа state → state для каждого типа синтаксических операторов. Для удобочитаемости мы используем λ-подобную конструкцию let (аналогичную конструкции let в языке ML). Элемент

```
let x ← a in body
```

используется для обозначения ((x) : body)(a) и имеет тот же смысл, что и λ-выражение

```
(λx.body a)
```

Если x принадлежит к типу области D → D', то выражение x[v/e] определяется как

```
x[v/e] = (D d) D' : if d = v then e else x(d)
```

Интуитивно понятно, что смысл этого выражения заключается в замене компонента v элемента x компонентом e и представляет собой базовую модель операции присваивания.

Семантика операторов

Теперь мы рассмотрим семантическое определение для каждого оператора нашего языка в терминах трансформаций (преобразований) состояний программы.

begin оператор end: последовательность ключевых слов begin ... end **не оказывает влияния на внутреннее состояние программы,** то есть она является истинно тождественной функцией в пространстве состояний. Это можно представить как

```
C{{begin stmt end}} = C{{stmt}}
```

composition (композиция): в данном случае нам хотелось бы применить к некоторому состоянию последовательно два оператора: $stmt_1$ и $stmt_2$. Мы можем представить это как суперпозицию функций:

```
C{{stmt1 ; stmt2}} = (state s)state : C{{stmt2}}(C{{stmt1}}(s))
```

Аргументом функции C{{$stmt_2$}} является состояние, получившееся в результате выполнения оператора $stmt_1$.

assignment (присваивание): оператор присваивания создает новое отображение памяти, используя значение, полученное при вычислении выражения exp в текущем состоянии:

```
C{{id ← exp}} = (state s)state : ((value v)state : s[id/v])(E{{exp}}(s))
```

if (ветвление): этот оператор определяет, какую из двух функций следует вычислить, причем сначала вычисляется значение функции E при заданном выражении в качестве аргумента, полученное значение передается в качестве аргумента булевой функции, а затем уже вычисляется соответственно оператор $stmt_1$ или оператор $stmt_2$:

```
C{{if exp then stmt1 else stmt2}} =
    (state s)state : ((bool b)state → state :
    (if b then C{{ stmt1}})else C{{stmt2}})(E{{exp}}(s))(s)
```

while (цикл): определение этого оператора неизбежно получается рекурсивным (rec), так как функция while сама является частью определения:

```
C{{while exp do stmt}} =
    rec(state s)state : ((bool b)state →state :
    (if b then C{{stmt}}∘C{{while exp do stmt}})
         else ((state s')state : s'))
    (E{{exp}}(s))(s)
```

Заметим, что в приведенном определении для C{{while}} мы действительно используем C{{while}} в его же определении (как часть выражения then). Это объяс-

няется членом rec, обозначающим рекурсию (recursive). Таким образом, наше определение while в действительности представляется в виде

```
C{{while}} = f(C{{while}})
```

Это утверждение можно сформулировать проще: оператор while является решением уравнения вида

$$x = f(x).$$

Такие решения называются *неподвижными точками*, и все, что нам нужно, — это наименьшая из всех неподвижных точек, которые решают это уравнение. Полный анализ теории неподвижных точек не входит в задачу авторов книги. Однако мы надеемся, что приведенные в этой главе материалы дают некоторое представление о том, как системы, подобные λ-исчислению, могут быть расширены для моделирования языков программирования.

4.2.3. Обзор языка ML

История. Язык ML (MetaLanguage) является аппликативным языком, программы на этом языке пишутся примерно так же, как на языке С или Pascal. Однако это аппликативный язык с улучшенной концепцией типов данных. ML поддерживает полиморфизм и, с помощью своей системы типов, абстракции данных. Основные структуры этого языка относительно компактны, особенно в сравнении с таким языком, как Ada. Но его возможность расширять типы данных обеспечивает ему большую мощь в случае написания сложных программ. ML включает создание и обработку исключительных ситуации (исключений), императивное и функциональное программирование, основанные на правилах спецификации, и большую часть концепций, представленных в других языках программирования. Если бы потребовалось выбрать один-единственный язык для изучения многих концепций языков программирования, то ML оказался бы наиболее подходящим кандидатом, пока не встал бы вопрос о коммерческой живучести.

ML завоевал большую популярность в исследовательских кругах и в области компьютерного образования. Доступность механизма определения типов данных на уровне исходной программы — то свойство ML, которое выгодно отличает его от других распространенных языков программирования. Однако коммерческих приложений, написанных на языке ML, практически нет; до сих пор он остается всего лишь инструментом при проведении теоретических исследований в области информатики и широко используется в образовании.

ML был разработан группой программистов во главе с Робертом Милнером (Robert Milner). Этот язык был задуман как механизм для построения (при помощи компьютера) формальных доказательств в системе логики для вычислимых функций, разработанной в середине 70-х гг. в Эдинбурге (Edinburgh Logic for Computable Functions). Но оказалось, что ML также полезен и в области символьных вычислений. В 1983 г. язык был пересмотрен, дополнен такими концепциями, как модули, и стал называться «стандартный ML» (Standard ML). Хотя обычно он реализуется как интерпретируемый язык, сравнительно легко можно создать и использовать его компилируемую версию. Первые компиляторы ML появились в 1984 г. [25].

К концу 80-х гг. Standard ML уже распространился в среде исследователей языков программирования. Одна из популярных версий ML была разработана Дэвидом Аппелем (David Appel) из Принстонского университета и Дэвидом Маккуином (David MacQueen) из AT&T Bell Telephone Laboratories. Именно эта версия использовалась для проверки примеров программ на ML, которые приведены в нашей книге.

Краткий обзор языка. ML — это сильно типизированный язык со статическим контролем типов и аппликативным выполнением программ. Отличие его от других представленных в этой книге языков заключается в том, что программист не должен объявлять типы данных — для этого существует специальный механизм вывода типа данных результирующих выражений. Этот механизм вывода типов делает возможным перегрузку и сопоставление с образцом на основе унификации, почти как в языке Prolog.

Как и в языке LISP, программа на ML состоит из определений нескольких функций. Каждая функция имеет статически определяемый тип и может возвращать значения любого типа. Поскольку ML является аппликативным языком, то хранение переменных в нем осуществляется иначе, чем в языках C или FORTRAN. В ML имеется только ограниченная форма присваивания. Функциональное выполнение программы на ML подразумевает, что параметры функций передаются по значению с созданием новой копии любого сложного объекта, используя кучу.

Комментарии в ML обозначаются следующим образом: (*...*). В отличие от многих других языков в ML комментарии могут быть вложенными. Как уже говорилось, в ML реализовано большинство свойств языков программирования, которые мы рассматриваем в этой книге. Он позволяет создавать записи и абстрактные типы данных, а также создавать и обрабатывать исключения. Возможности ввода-вывода в ML не очень велики, что обусловлено областью его применения — большинству исследователей не требуется обрабатывать многочисленные базы данных с использованием сложных форматов. Синтаксис этого языка достаточно лаконичен по сравнению с синтаксисом C++ или Ada.

4.2.4. Проверка правильности программы

При разработке и написании программ все большее значение придается их корректности и надежности. Современные языки имеют в своем арсенале инструменты для улучшения этих характеристик создаваемых программ. Некоторые концепции, упоминавшиеся нами при обсуждении семантики языка, могут помочь найти ответы на три вопроса, имеющих отношение к корректности программы.

1. Дана программа P, каков ее смысл? Иными словами, какова спецификация S этой программы?
2. Дана спецификация S, как разработать программу P, которая реализует эту спецификацию?
3. Выполняют ли одну и ту же функцию программа P и ее спецификация S?

Первый из этих вопросов непосредственно связан с моделированием семантики языка программирования, рассмотренным в разделе 4.2.2. Второй вопрос сводится к задаче построения хорошей программы по заданной спецификации (на

сегодняшний день это центральная задача в технологиях разработки программного обеспечения). Третий вопрос формулирует основную проблему проверки правильности программы, то есть соответствия программы ее спецификации. Хотя эти три вопроса различаются по форме, они одинаковы по существу.

Третья формулировка была популярна в 70-е гг., первая — с середины 70-х до конца 80-х, а вторая представляет большой интерес для современных исследователей. Однако ответы на все три вопроса следует искать с помощью схожих методов. Тестирование программы не может гарантировать отсутствие ошибок, за исключением лишь очень простых программ. При тестировании программа выполняется несколько раз, причем для каждого раза входные данные берутся из специального тестового набора. Результат выполнения программы сравнивается с входными данными. Если результаты достаточно большой серии проверок неизменно оказываются правильными, то программа называется отлаженной. Но на самом деле мы знаем только то, что программа корректно работает с тестовыми входными данными. В других случаях программа может ошибаться. Но перебрать все допустимые входные данные, как правило, невозможно. Таким образом, при проверке программ мы должны удовлетвориться неполной гарантией ее корректности. Если бы существовал какой-либо метод проверки правильности программы, не основанный на тестировании, то программы стали бы более надежными.

Программа всегда вычисляет какую-нибудь функцию. Программист, разумеется, знает, какую функцию программа *должна* вычислять. Предположим, что в программу отдельно встроена спецификация этой функции. Предположим также, что по виду программы мы в состоянии определить, какую функцию она *вычислила на самом деле*. Если бы мы могли показать, что программа в действительности вычисляет точно определенную функцию, то и без тестирования это было бы достаточным доказательством корректности программы.

Исчисление предикатов — это система обозначений в формальной логике, которая полезна, в частности, для точного определения сложных функций. Разработано несколько подходов к доказательству корректности программ, основанных на предположении, что функция, вычисляемая программой, определена с помощью исчисления предикатов. Программа исследуется на предмет соответствия вычисляемой ею функции той, которая задана с помощью формулы исчисления предикатов.

Обычно используется следующее обозначение: $\{P\}S\{Q\}$. Смысл его в том, что если предикат P истинен до начала выполнения оператора S и если последний выполняется и завершается, то после выполнения оператора S истинным будет и предикат Q. Дополнив это обозначение несколькими правилами вывода, мы можем встроить эту структуру в систему доказательств на основе исчисления предикатов, как показано в следующей таблице.

Правило	**Предпосылка**	**Следствие**
1. Консеквенция$_1$	$\{P\}S\{Q\}.\ (Q \Rightarrow R)$	$\{P\}S\{R\}$
2. Консеквенция$_2$	$(R \Rightarrow P).\ \{P\}S\{Q\}$	$\{R\}S\{Q\}$
3. Композиция	$\{P\}S_1\{Q\}.\ \{Q\}S_2\{R\}$	$\{P\}S_1;\ S_2\{R\}$

Правило	Предпосылка	Следствие
4. Присваивание	x:=expr	{P(expr)}x := expr{P(x)}
5. Условие	$\{P \wedge B\}S_1\{Q\}, \{P \wedge \neg B\}S_2\{Q\}$	{P} **if** B **then** S_1 **else** S_2{Q}
6. Цикл while	$\{P \wedge B\}S\{P\}$	{P} **while** B **do** $S\{P \wedge \neg B\}$

Если удастся показать, что предпосылка в данном правиле вывода истинна, то можно заменить ее следствием. Это позволяет строить доказательства правильности программ в пределах известной нам концепции исчисления предикатов.

Аксиоматическая верификация обычно проводится в обратном порядке по отношению к ходу выполнения программы. Зная результат выполнения какого-либо оператора (постусловие), мы можем вывести, какое предусловие должно быть истинно, до начала выполнения этого оператора. Например, если постусловием для оператора присваивания X := Y + Z является X > 0, то, согласно аксиоме присваивания:

{P(expr)}x := expr{P(x)}

где P(x) = X > 0, мы получаем

{Y+Z>0}X := Y + Z{X > 0}.

Таким образом, предусловием для нашего оператора присваивания должно быть Y + Z > 0. Продолжая в том же духе, мы можем доказать корректность (но уже не с такой легкостью) последовательности операторов, содержащей операторы присваивания, условия (if) и цикла (while).

Возможно, наиболее интересным является правило вывода для оператора цикла while:

if $\{P \wedge B\}S\{P\}$ then {P} while B do $S\{P \wedge \neg B\}$.

Предикат P обычно называется *инвариантом*, он должен быть истинным и до начала цикла while, и по его окончании. Хотя для отыскания инвариантов существуют эвристические методы, но в целом эта проблема не имеет решения (см. раздел 4.1.3). Тем не менее для многих корректных программ инварианты можно определить.

Листинг 4.1. Программа для вычисления y = A × B

```
      {B≥0}
1.  MULT(A,B) ≡ {
2.    a:=A;
3.    b:=B
4.    y:=0;
5.    while b>0 do
      begin
6.       y:=y+a;
7.       b:=b-1;
         end}
    {y = A×B}
```

В примере 4.1 содержится доказательство правильности программы MULT, которая вычисляет **y = A × B**. При изучении программы можно увидеть, что на каждом шаге цикла while к y добавляется a, в то время как произведение a × b **уменьшается на** a **за счет уменьшения** b на 1. Следовательно, в данном случае инвариантом является $y + a \times b = A \times B \vee b \geq 0$.

Условие {P}S{Q} означает, что если S завершится и P истинно перед началом выполнения S, то Q будет истинным по окончании S. Для оператора присваивания и условного оператора `if` здесь не возникает проблем. Оба они должны завершится. Но для оператора `while` это не всегда так: нужно еще доказать, что цикл завершится. Обычно используется следующий метод:

1. Сначала следует показать, что существует некая целочисленная функция f, такая что в процессе выполнения цикла эта функция остается положительной, $f>0$.
2. Если f_i — это значение функции f во время i-го выполнения цикла, то $f_{i+1} < f_i$.

Если оба условия выполняются, то такой цикл должен завершиться (см. задачу 8 в конце этой главы).

Рассмотренная система будет полезной на практике только в том случае, если удастся автоматизировать процесс доказательства корректности программы; применение к реальным программам методов доказательства, предложенных в этой главе, требует массы достаточно утомительных вычислений. Ошибка в доказательстве может привести к выводу, что программа корректна, в то время как это, возможно, не так. По этим причинам желательно автоматизировать процесс доказательства таким образом, чтобы на основе функциональной спецификации программы и самой программы автоматическая система доказательства могла без участия (или с минимальным участием) программиста доказать ее корректность.

В настоящее время методы доказательства правильности программ считаются полезными, если их можно применять уже на стадии проектирования программы, то есть чтобы программу можно было исследовать на корректность по мере ее написания. Оказалось, что если программа была написана обычным образом, то есть не структурирована специально для облегчения проверки корректности, ее очень трудно или даже невозможно проверять автоматически. Методы автоматической проверки корректности иногда бывают полезны, но они являются недостаточно мощными, чтобы взаимодействовать с чрезвычайно сложными структурами, которые встречаются в более старых, неоднократно модифицированных программах. Методы доказательства корректности программ обычно преподаются программистам-практикам, а также рассматриваются в различных вводных курсах по программированию.

Исследования в этой области повлияли на идеологию новых языков программирования. При возможности выбора разработчики языка отдают предпочтение тем свойствам, которые не препятствуют применению методов доказательства правильности программ, и пытаются избегать тех свойств, которые затрудняют это доказательство. Например, были проведены исследования, показавшие, что следует избегать включения в язык возможности создания *псевдонимов имен*, так как это препятствует доказательству корректности программ на этом языке (см. раздел 9.2.1). В некоторые языки, например в C++, включен специальный макрос `assert`, возможности которого основаны отчасти на описанном аксиоматическом методе (см. раздел 11.1.1). С помощью этого макроса в текст исходной программы вставляется некое утверждение, которое «тестируется» во время выполнения программы. Подобный подход не является *априорным* методом доказательства корректности программы, но вставка утверждения в текст программы позволяет обнаружить множество ошибок, которые проявляются при дальнейшем ее использовании.

Пример 4.1. Аксиоматическое доказательство правильности программы MULT

Обычно доказательство происходит в обратном порядке по отношению к ходу выполнения программы, то есть предпосылка выводится из следствия. Нам необходимо найти инварианты для цикла `while` (строки 5–7 листинга 4.1): y увеличивается на 1, в то время как b уменьшается на 1. Тогда инвариантом является

$$(y + ab = AB) \wedge (b \geq 0)$$

	Оператор	Основание
a	$\{y + a(b - 1) = AB \wedge (b - 1) \geq 0\}$ $b := b - 1\{y + ab = AB \wedge b \geq 0)\}$	Правило присваивания (строка 7)
b	$\{y + ab = AB \wedge b - 1 \geq 0\}y := y + a$ $\{y + a(b - 1) = AB \wedge b - 1 \geq 0\}$	Правило присваивания (строка 6)
c	$\{y + ab = AB \wedge b - 1 \geq 0\}$ $y := y + a;\ b := b - 1$ $\{y + ab = AB \wedge b \geq 0\}$	Правило композиции (a, b)
d	$(y + ab = AB) \wedge (b \geq 0) \wedge (b > 0) \Rightarrow$ $(y + ab = AB) \wedge b - 1 \geq 0$	Теорема
e	$\{y + ab = AB \wedge (b \geq 0) \wedge (b > 0)\}$ $y := y + a;\ b := b - 1$ $\{y + ab = AB \wedge b \geq 0\}$	Правило консеквенции$_2$ (c, d)
f	$\{y + ab = AB \wedge (b \geq 0)\}$**while**... $\{y + ab = AB \wedge b \geq 0 \wedge \neg b > 0\}$	Правило цикла `while` (строка 5, e)
g	$\{0 + ab = AB \wedge b \geq 0\}$ $y := 0\{y + ab = AB \wedge b \geq 0\}$	Правило присваивания (строка 4, f)
h	$\{0 + aB = AB \wedge B \geq 0\}\ b := B$ $\{0 + ab = AB \wedge b \geq 0\}$	Правило присваивания (строка 3, g)
i	$\{0 + AB = AB \wedge (B \geq 0)\}a := A$ $\{0 + aB = AB \wedge B \geq 0\}$	Правило присваивания (строка 2, h)
j	$B \geq 0 \Rightarrow 0 + AB = AB \wedge B \geq 0$	Теорема
k	$\{B \geq 0\}a := A\{0 + AB = AB \wedge B \geq 0\}$	Правило консеквенции$_2$
l	$\{B \geq 0\}\ a := A;\ b := B;\ y := 0$ $\{y + ab = AB \wedge b \geq 0\}$	Правило композиции (k, h, g)
m	$\{B \geq 0\}$ MULT(A,B) $\{y + ab = AB \wedge b \geq 0 \wedge \neg b > 0\}$	Правило композиции (l, f)
n	$(y + ab = AB) \wedge (b \geq 0) \wedge \neg(b > 0) \Rightarrow$ $(b = 0) \wedge (y = AB)$	Теорема
o	$\{B \geq 0\}$ MULT(A,B) $\{y = AB\}$	Правило консеквенции$_1$ (m, n)

Также нужно показать, что программа завершит свое выполнение. Достаточно показать, что должны завершиться все циклы `while`. Обычный способ таков.

1. Показать, что существует некоторое свойство, которое всегда положительно во время выполнения цикла (например, $b \geq 0$).
2. Показать, что при последовательных итерациях цикла это свойство уменьшается (например, $b := b - 1$).

Если оба свойства остаются истинными, то цикл должен завершиться.

4.2.5. Алгебраические типы данных

Операции перезаписи членов и унификации, описанные нами ранее, играют важную роль в развитии модели алгебраических типов данных. Если мы описываем отношения между рядами функций, то мы констатируем, что любая реализация, согласованная с этими отношениями, является корректной реализацией.

Например, стек (stack), содержащий целые числа, можно определить с помощью следующих операций:

```
push :       stack × integer → stack
pop :        stack → stack
top :        stack → integer ∪ {undefined}
empty :      stack → Boolean
size :       stack → integer
newstack :   → stack
```

Операции push и pop здесь используются в своей обычной интерпретации, top возвращает верхний элемент стека, не удаляя его, empty проверяет, не является ли стек пустым, size возвращает количество элементов в стеке и newstack создает новый экземпляр стека.

Генерирование алгебраических аксиом

Пусть существует множество операций, определяющих тип данных. Мы можем привести некоторые эвристические соображения для построения взаимоотношений между этими операциями. Операции разделяются на три класса: генераторы, конструкторы и функции.

Генераторы. Для абстрактного типа данных x генератор g создает новый экземпляр этого типа. Следовательно, его сигнатурой будет

```
g : not_x → x
```

(not_x обозначает любой тип данных, отличный от x). В нашем примере со стеком генератором является операция newstack.

Конструкторы. Конструкторы c модифицируют экземпляры абстрактного типа x и имеют следующую сигнатуру:

```
c : x × not_x → x
```

Примером конструктора является операция push.

Функции. Все остальные операции над данными абстрактного типа x относятся к функциям. В нашем примере со стеком функциями являются top, pop, size и empty.

Интуитивно понятно, что любой объект y абстрактного типа x создается посредством применения генератора и повторного применения конструкторов. Для стеков это означает, что любой стек является результатом использования генератора newstack, который создает пустой стек, и последовательного применения оператора push для помещения в стек каких-либо объектов. Например, стек, содержащий цифры 1, 5, 3, может быть получен следующим образом:

```
push(push(push(newstack. 1). 5). 3)
```

Хотя не существует формальной модели разработки алгебраических аксиом, с успехом применяется следующий эвристический подход. Генерируется аксиома для каждой функции с каждым конструктором и генератором. Поскольку в нашем примере со стеком имеются три функции, один конструктор и один генератор, то

у нас получается шесть аксиом. Если уже написана левая часть выражения для аксиомы, ее значение (правую часть выражения) обычно легко определить. Например, для функции top и конструктора push аксиома выглядит следующим образом:

```
top(push(S, I)) = ...
```

Интуитивно ясно, что верхним элементом стека должен быть тот, который только что добавлен, поэтому аксиома получается такая:

```
top(push(S, I)) = I
```

Продолжая в том же духе, мы получим следующие аксиомы:

```
1: pop(newstack) =      newstack
2: pop(push(S, I) =     S
3: top(newstack) =      undefined
4: top(push(S, I) =     I
5: empty(newstack) =    true
6: empty(push(S, I) =   false
7: size(newstack) =     0
8: size(push(S, I) =    size(S) + 1
```

Эти аксиомы часто называются правилами перезаписи, при помощи которых можно перезаписать экземпляр какого-либо объекта в более простом виде. Согласно приведенной спецификации, операция push возвращает экземпляр стека (stack), сконструированного из уже имеющегося стека и некоторого целого числа. Например, запрос может быть сформирован как выполнение операции newstack, затем push и еще раз push, затем pop и, наконец, выполнение функции empty:

```
empty(pop(push(push(newstack, 42), 17)))
```

Унификация этого выражения при помощи аксиомы 2 позволяет упростить его за счет сокращения pop и push — противоположных по своему действию операций, следующих друг за другом. Тогда выражение принимает следующий вид:

```
empty(push(newstack, 42)))
```

Теперь мы можем использовать аксиому 6 и упростить это выражение до false. Нам удалось модифицировать выражение таким образом, что функция empty применяется только к суперпозиции конструктора и генератора и выдает значение false. Это является иллюстрацией к нашему рассказу о конструкторах. Алгебраическое выражение, состоящее только из генераторов и конструкторов, называется *канонической*, или *нормальной*, формой выражения.

Индукция типов данных

Пусть дан набор аксиом (то есть отношений) между множеством операций алгебраического типа данных. Часто требуется проверить истинность определенных свойств. Затем эти свойства становятся обязательными для всех программ (например, на C или на Pascal), реализующих данную спецификацию. *Индукция типов данных* — это один из способов проверки подобных свойств. Он тесно связан с методом математической индукции.

Пусть P(y) — некий предикат, связанный с y ∈ x. При каких условиях P всегда будет истинен для любых членов типа x? Как и в случае математической индукции, можно доказать, что P будет истинно для любых элементарных объектов этого типа[1].

[1] Аналог базы математической индукции. — *Примеч. науч. ред.*

Затем мы можем показать, что при конструировании новых объектов того же типа из элементарных наше свойство P по-прежнему остается истинным. /

Индукция типов данных определяется следующим образом.

1. Имеется тип данных x с генераторами f_i, конструкторами g_i, другими функциями h_i, а также предикат P(y) для y ∈ x.
2. Нужно показать, что $P(f_i)$ истинно. Это является базой индукции для доказательства — значение предиката на функциях-генераторах истинно.
3. Предположим, что P(y) истинно. Нужно показать, что из этого следует истинность $P(g_i(y))$. Тогда можно расширить предикат P на все объекты, созданные посредством применения генератора и последующего применения конструкторов.
4. Теперь можно сделать вывод, что P(y) истинно для всех y, принадлежащих типу x.

Для нашего примера со стеком нужно показать, что предикаты P(newstack) и P(push(x, i)) истинны. Используя индукцию типов данных, можно показать, что добавление в стек нового объекта увеличивает размер стека, то есть

```
size(push(S, X)) > size(S)
```

Доказательство этого утверждения вы найдете в примере 4.2.

ПРИМЕР 4.2. Индукция типов данных

1. Для доказательства истинности предиката P(S) > size(push(S,X)) > size(S) мы преобразуем его при помощи аксиом, которые будут применяться до тех пор, пока дальнейшее его упрощение не станет невозможно:

   ```
   size(push(S, X)) > size(S) – предположение теоремы
   size(S) + 1 > size(S) – по аксиоме 8
   ```

2. Теперь мы применяем индукцию типов данных для того, чтобы доказать справедливость size(S) + 1 > size(S) для генератора newstack и конструктора push, то есть нам нужно доказать истинность P(newstack) и P(push(S, i)).

   ✦ *База индукции.* Заменим S на newstack и покажем истинность нашего предиката:

   ```
   0 + 1 > 0 + 0 – простой математический факт
   size(newstack) + 1 > size(newstack) – по аксиоме 7
   ```

   Итак, для S = newstack теорема доказана и таким образом построена база для индукции.

   ✦ *Индукционный переход.* Предположим, что для S теорема выполняется, то есть size(S) + 1 > size(S). Покажем, что тогда она выполняется и для S' = push(S, X):

   ```
   size(S) + 1 > size(S) – индукционное предположение
   size(S) + 1 + 1 > size(S) + 1 – добавим 1 к обеим частям неравенства
   size(push(S, X)) + 1 > size(push(S, X)) – по аксиоме 8
   ```

Итак, индукционный переход также выполняется.

Вывод. Мы доказали, что

```
size(push(S, X)) > size(S)
```

выполняется для всех стеков S.

4.3. Рекомендуемая литература

Формальные модели синтаксиса, синтаксического разбора и трансляции широко используются при анализе и реализации компиляторов и трансляторов (см. ссылки в конце главы 3). В работах Стансифера (Stansifer) [104] подробно обсуждается применение λ-исчисления и денотационной семантики при разработке языков программирования.

Обзор методов формального определения семантики можно найти в работах Маркотти и др. (Marcotty) [78]; там же вы найдете описание некоторых методов. В статье Хоара и Лауера (Lauer) [52] приведен сравнительный анализ нескольких методов определения семантики простого языка. Лукас (Lucas) и Уолк (Walk) [75] описали операционный метод, называемый *Vienna Definition Language*, который был использован для формального определения семантики языка PL/I. В [5] представлена большая часть формального определения семантики Ada; аксиоматическое определение большинства конструкций языка Pascal дано в работе Хоара [51].

Обзор методов верификации программ вы найдете в [123] и книге Ганнона (Gannon), Пуртило (Purtilo) и Зелковица (Zelkowitz) [43]. Алгебраические типы данных исследовались Гуттагом (Guttag) [48]. Метод формальной спецификации *Vienna Development Method* (VDM) описан в книге [61], а метод Z — в статье [103]. В этой главе мы смогли коснуться лишь некоторых из многих изящных теоретических моделей и результатов исследований в области семантики программ, универсальных языков и абстрактных машин типа машин Тьюринга.

4.4. Задачи и упражнения

1. Дополните грамматику, приведенную в табл. 3.1, следующими НФБ-правилами:

```
<программа> ::=          <объявление> <список операторов>
<объявление> ::=         <decl> <объявление> | <decl>
<decl> ::=               declare <переменная>;
<список операторов> ::=  <оператор присваивания> <список операторов> |
                            <оператор присваивания>
```

   Разработайте атрибутивную грамматику, для которой:

   - оператор присваивания определен корректно в том и только в том случае, если каждая переменная в этом операторе объявлена;
   - программа определена корректно, если все операторы присваивания в этой программе определены корректно.

2. Покажите, что язык, сгенерированный следующей грамматикой, является регулярным языком:

   $S \to aSa|a$

3. Палиндром — это строка, которая читается в обе стороны одинаково.

   - Покажите, что множество палиндромов нечетной длины, составленных из символов {a, b}, является констекстно-свободным языком.

- Покажите, что множество строк нечетной длины, составленных из символов {a, b}, *отличных* от палиндромов, также является констекстно-свободным языком.
- Почему для распознавания множества палиндромов нечетной длины требуется недетерминированный автомат с магазинной памятью, а детерминированный не годится?

4. Пусть имеется контекстно-свободная грамматика

```
S ::= 0S0 | 1s1 | 0 | 1
```

Изобразите дерево вывода для 0110110.

5. Мы знаем, что для генерации строк типа $a^n b^n$ требуется контекстно-свободная грамматика. Рассмотрим следующую регулярную грамматику:

```
S :: = aS | bS | a | b
```

Утверждается, что эта грамматика также генерирует $a^n b^n$. Например, $a^3 b^3$ генерируется следующим образом:

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaabS \Rightarrow aaabbS \Rightarrow aaabbb$

В данном случае мы видим, что при помощи грамматики типа 3 сгенерирован язык типа 2. Объясните это противоречие.

6. Модифицируйте все процедуры из листинга 3.1 так, чтобы получить постфиксную запись для арифметических выражений.
7. Может ли регулярная грамматика распознавать определенный ниже язык для подмножества выражений? Приведите объяснение. Объясните также основное различие между этой грамматикой и грамматикой всех возможных выражений

```
E → E + T | T
T → T * P | P
P → i
```

8. С помощью аксиоматического метода доказательства корректности программ:
   - покажите, что приведенные в тексте главы два условия окончания цикла while действительно гарантируют его завершение (подсказка: что произойдет, если оба условия истинны, а цикл не завершается?);
   - почему требуется, чтобы функция f была целочисленной?
9. Докажите правильность следующей программы целочисленного деления:

```
{x ≥ 0 ∧ y > 0}
q := 0
r := 0
while y ≤ r do
      begin
      r := r - y:
      q := q + 1
      end
{y > r ∧ x = r + y × q}
```

10. Что произойдет, если в листинге 4.1 окажется, что B меньше 0? Для какой области значений A и B программа гарантированно завершается и дает правильный результат?

11. Используя определения для логических not, and и or через λ-выражения, покажите, что:
    - ((not A) B) = истина тогда и только тогда, когда A = истина и B = истина;
    - ((or A) B) = истина тогда и только тогда, когда A = истина или B = истина.

    То есть нужно показать, что определения функций and и or через λ-выражения соответствуют обычной интерпретации этих логических операций.
12. Используя определения целых чисел через λ-выражения, покажите, что $2 + 3 = 5$.
13. В тексте главы в том примере, где стеки иллюстрируют алгебраические типы данных, предполагается, что стеки неограниченны. Конструктор push можно применять к любому стеку. Но реально длина любого стека конечна. Перепишите аксиомы для стеков для случая, когда существует некий максимальный размер стека MaxStack.
14. Создайте аксиомы для других типов данных, например для очередей и множеств. Какие нужны операции для определения этих типов?
15. Реализуйте стеки, очереди или множества в языке C++ и покажите, что в вашей реализации выполнены все соответствующие аксиомы.
16. Рассмотрим следующие аксиомы сложения (succ означает successor, то есть функцию, добавляющую единицу к предыдущему значению):

    ```
    add(0,X) = X
    add(succ(X),Y) = succ(add(X,Y))
    ```

    Используя только эти две аксиомы, докажите при помощи индукции типов данных, что

    ```
    add(X,Y) = add(Y,X)
    ```

    и

    ```
    add(add(X,Y),Z) = add(X,add(Y,Z))
    ```
17. Рассмотрим аксиомы для стеков из раздела 4.2.5. Если интерпретировать их как правила языка Prolog, то многие алгебраические свойства можно реализовать как программы на этом языке. Разработайте набор правил языка Prolog, преобразующих последовательность операций push и pop в более простую последовательность, состоящую только из операций push (то есть речь идет о том, чтобы привести эту последовательность к нормальной форме). Например, pop(push(S,I)) приводится к S, а push(pop(push(pop(push(S, I)), J)), K) преобразуется в push(S, K).
18. Покажите эквивалентность следующих объектов:
    - конечного автомата и регулярных языков;
    - автомата с магазинной памятью и контекстно-свободных языков;
    - линейно-ограниченного автомата и контекстно-зависимых языков. Линейно-ограниченный автомат — это машина Тьюринга, в которой лента содержит только входные данные (следовательно, ее длина возрастает соответственно количеству вводимых данных, хоть и не может быть неограниченной);
    - машины Тьюринга и языков с фразовой структурой.

19. Покажите эквивалентность или ее отсутствие между детерминированной и недетерминированной моделями:
    - для конечного автомата;
    - для автомата с магазинной памятью;
    - для линейно-ограниченного автомата;
    - для машины Тьюринга.

    Первая и последняя задачи сравнительно простые, вторая несколько труднее. Тем же, кто решит третью задачу, обеспечено немедленное получение докторской степени и широкая известность в академических кругах. Эта не решенная до сих пор задача связана с задачей НП-полноты (см. раздел 4.1.3).

Глава 5. Элементарные типы данных

Любая программа, в сущности, представляет собой набор операций, которые применяются к определенным данным в определенной последовательности. Основные различия между языками программирования сводятся к тому, каковы допустимые типы данных и операций и какие механизмы используются для управления последовательностью применения операций к данным. Эти три области — данные, операции и механизмы управления — составляют основу для сравнения языков программирования, которые обсуждаются в нашей книге. В этой главе мы рассмотрим данные, типы и операции, которые обычно встроены непосредственно в язык; в главе 6 обсуждается понятие *абстракции* — так называется расширение встроенных типов данных за счет добавления новых типов, определяемых самим программистом.

5.1. Свойства типов и объектов

Сначала мы исследуем свойства, которые определяют объекты данных в языках программирования. Затем мы обсудим стандартные для большинства языков типы данных с точки зрения способов их реализации в аппаратной части компьютера (то есть элементарные типы данных). В главе 6 мы рассмотрим такие типы данных, которые обычно моделируются при помощи программного обеспечения (то есть структурные типы данных).

5.1.1. Объекты данных, переменные и константы

Области хранения данных в аппаратной части компьютера (память, регистры и внешние запоминающие устройства) обычно имеют довольно простую структуру в виде последовательности битов, сгруппированных в байты или слова. Однако хранение данных в виртуальном компьютере, как правило, организовано более сложным образом — в различные моменты выполнения программы используются такие формы хранения данных, как стеки, массивы, числа, символьные строки и некоторые другие. Один или несколько однотипных элементов данных, объединенных в одно целое в виртуальном компьютере в некоторый момент выполнения программы, принято называть *объектом данных*. В течение времени выполнения программы существует множество объектов данных различных типов. Более того, в отличие от сравнительно неизменной статической организации хранения данных в аппа-

ратной части компьютера объекты данных и отношения между ними динамически меняются в процессе выполнения программы.

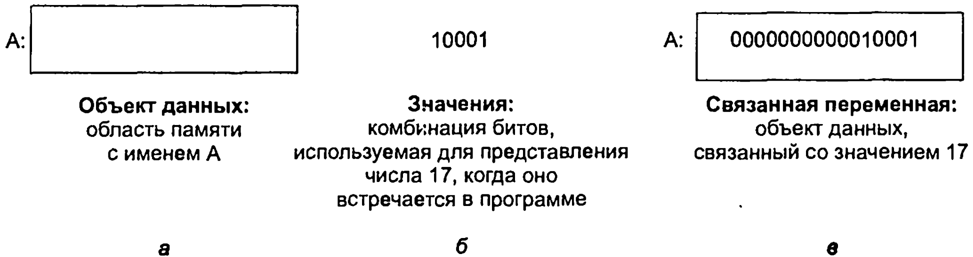


Рис. 5.1. Пример объекта данных — простая переменная со значением 17

Некоторые объекты данных *определяются программистом* (например, переменные, константы, массивы, файлы и т. д.); программист создает эти объекты и управляет ими явным образом при помощи объявлений и операторов в программе. Другие объекты данных *определяются системой* (то есть автоматически создаются по мере необходимости виртуальным компьютером во время работы программы как вспомогательные, предназначенные для выполнения служебных функций). К таким объектам данных (например, к стекам, записям активации подпрограмм, файловым буферам и спискам свободной памяти) у программиста нет непосредственного доступа.

Объект данных представляет собой *контейнер для хранения значений данных* — то есть то место, где эти значения хранятся и откуда они затем извлекаются. Объект данных характеризуется набором *атрибутов*, самым важным из которых является *тип данных*. Атрибуты определяют количество и тип значений, которые могут содержаться в объекте данных, а также определяют логическую организацию этих значений.

Значение данных может представлять собой отдельное число, символ или указатель на другой объект данных. Значение обычно представлено конкретной комбинацией битов в памяти компьютера. До сих пор мы предполагали, что два значения совпадают, если представляющие их в памяти компьютера комбинации битов идентичны. Но для более сложных типов данных такого простого определения недостаточно. Различие между объектами и значениями данных во многих языках недостаточно отчетливо и их довольно легко перепутать. Пожалуй, наиболее ярко это различие проявляется в реализации: объект данных представлен некоторой областью в памяти компьютера, а значение данных представлено комбинацией битов. Если мы говорим, что объект данных A содержит значение B, то имеется в виду следующее: в области памяти компьютера, представляющей объект A, расположена определенная комбинация битов, соответствующая значению B (рис. 5.1).

Если мы пронаблюдаем за работой программы, мы заметим, что одни объекты данных существуют с самого начала выполнения программы, а другие создаются динамически в процессе ее выполнения. Одни объекты данных в какой-то момент уничтожаются, а другие сохраняются до конца работы программы. Таким образом, у каждого объекта данных имеется определенное время жизни, в тече-

ние которого он может быть использован для хранения значений данных. Объект данных называется *элементарным*, если содержащееся в нем значение всегда фигурирует в программе как единое целое. Если же этот объект представляет собой совокупность некоторых других объектов, то он называется *структурой данных*.

Любой объект данных за время своей жизни участвует в нескольких связываниях. Хотя атрибуты объекта данных инвариантны в течение его времени жизни, связывания могут динамически изменяться. Ниже перечислены наиболее важные атрибуты и связывания.

1. *Тип*. Этот атрибут ассоциирует объект данных со множеством значений, которые могут содержаться в этом объекте данных.
2. *Местоположение*. Связывание объекта данных с определенным местоположением (областью памяти) обычно задается и контролируется специальными вспомогательными программами управления памятью, которые предусмотрены в виртуальном компьютере и недоступны непосредственно для программиста.
3. *Значение*. Это связывание обычно является результатом операции присваивания.
4. *Имя*. Связывание объекта данных с одним или несколькими именами, по которым к нему происходят обращения во время работы программы, обычно осуществляется при помощи объявлений и может модифицироваться при входе и выходе из подпрограмм (подробнее см. главу 9).
5. *Компонент*. Связывание объекта данных с одним или более объектами данных, компонентом которых он является, часто представлено значением указателя и может быть изменено посредством изменения этого указателя (см. раздел 5.3.2).

Переменные и константы

Переменная — это объект данных, который явным образом определен и назван программистом. *Простая переменная* — это элементарный объект данных, имеющий имя. Обычно мы представляем себе переменную как объект, значение (или значения) которого можно менять посредством операции присваивания (то есть связывание переменной с ее значением может изменяться несколько раз за время жизни переменной). Если не имеет значения, заглавные или строчные буквы используются в именах переменных (то есть если имена, например `MYVARIABLE` и `myvariable`, воспринимаются программой как обозначающие один и тот же объект), то говорят, что имена переменных *нечувствительны к регистру*. Если же эти имена соответствуют различным объектам, то говорят, что имена *чувствительны к регистру*.

Константа — это объект данных, имя которого неизменно связано со значением (или значениями) в течение всего времени жизни. *Буквальная константа* (или *литерал*) — это константа, имя которой является просто формой записи ее значения (например, «21» — это десятичная форма записи буквальной константы, которая представляет собой объект данных со значением 21). *Определяемая программистом*, или *именованная*, константа — это объект данных, имя

для которого программист выбирает произвольным образом при его определении.

Пример 5.1. Простые переменные в языке С

Подпрограмма на языке С может содержать следующее объявление:

```
int N:
```

что означает объявление простого объекта данных N целого типа (integer). Далее в подпрограмме может встретиться операция присваивания

```
N = 27:
```

используемая для присвоения значения 27 объекту данных с именем N. Более полно эту ситуацию можно описать так.

1. Посредством объявления создается элементарный объект данных целочисленного типа.
2. Этот объект данных должен быть создан при входе в подпрограмму и уничтожен при выходе из нее; таким образом, время его жизни равно времени выполнения подпрограммы.
3. В течение времени своей жизни этот объект данных связан с именем N, по которому к нему можно обращаться в подпрограмме, как, например, в приведенной выше операции присваивания. С ним могут быть связаны другие имена, если он передается как параметр в другую подпрограмму.
4. Изначально с объектом данных не связано никакое значение, но оператор присваивания временно связывает этот объект данных со значением 27, пока последующий оператор присваивания значения переменной N не изменит текущее связывание.
5. От программиста остаются скрытыми другие связывания, которые происходят в виртуальном компьютере: объект данных N может стать компонентом *записи активации*, то есть объекта данных, в котором содержатся все локальные данные для подпрограммы, а под эту запись активации отводится определенное место в создаваемом в момент начала выполнения подпрограммы стеке (еще один скрытый от программиста объект). Когда подпрограмма завершается, выделенная под стек область памяти освобождается для дальнейшего использования и связывание объекта данных с областью памяти разрушается (более подробно эта тема обсуждается в главе 6).

Поскольку в случае с константой связывание значения с ее именем остается неизменным в течение всего времени жизни этой константы, информация об этом связывании доступна транслятору. Следовательно, если программист создает программу на языке С и пишет: #define MAX 30, то эта информация известна уже во время трансляции. Компилятор языка С может извлечь из такой информации определенную пользу — например, если в программе встретится оператор присваивания MAX = 4, это будет воспринято как ошибка, поскольку значение константы не должно меняться, присваивание константе MAX значения 4 имеет столько же смысла, что и оператор присваивания 30 = 4. Иногда компилятор может использовать информацию о значении константы для того, чтобы избежать генерирования кода для некоторого выражения или оператора. Например, в условном операторе if

```
if (MAX < 2) {…}
```

уже содержится информация для транслятора о реальном значении именованной константы MAX и буквальной константы 2, следовательно, транслятор может вычислить, что булево выражение MAX < 2 ложно, и проигнорировать полностью весь код для этого условного оператора if.

Пример 5.2. Переменные, константы и литералы в языке С

В подпрограмму на языке С могут входить следующие объявления:

```
const int MAX=30;
int N;
```

Затем мы можем написать следующие операторы присваивания:

```
N = 27;
N = N + MAX;
```

N — это простая переменная, а MAX, 27 и 30 — константы. N, MAX, 27 и 30 — это имена объектов данных целого типа. Объявление константы определяет, что во время выполнения подпрограммы объект данных, названный MAX, постоянно должен быть связан со значением 30. Константа MAX — это *определенная программистом константа*, так как программист явным образом определил имя для значения 30. С другой стороны, имя 27 является *литералом*, который именует объект данных, содержащий значение 27. Такие литералы являются частью определения самого языка программирования. Важно понимать тонкое отличие между *значением* 27, которое является целым числом, представленным в памяти компьютера во время выполнения программы последовательностью битов, и *именем* «27», которое состоит из двух символов, 2 и 7, и является десятичной формой записи в тексте программы того же числа. В языке С предусмотрены как объявления констант, подобные приведенным в этом примере, так и макроопределения типа #define MAX 30, которое представляет собой операцию, выполняемую во время компиляции и приводящую к тому, что все ссылки на имя MAX в подпрограмме будут заменены на константу 30.

Обратите внимание на то, что в приведенном примере у константы 30 имеются два имени: определенное программистом имя MAX и буквальное имя 30; оба этих имени могут быть использованы в программе для ссылки на один и тот же объект данных со значением 30.

Следует также учитывать, что

```
#define MAX 30
```

является командой, которая используется транслятором для того, чтобы приравнять MAX значению 30, в то время как атрибут const в языке С является указанием транслятору, что переменная MAX всегда содержит значение 30.

Сохраняемость. Большинство программ до сих пор создается с использованием пакетной модели обработки (см. раздел 1.2.2). То есть программист предполагает следующую последовательность действий:

1) программа загружается в память;
2) необходимые данные, расположенные на внешних носителях информации (например, на дисках, магнитных лентах и т. п.), становятся доступными программе;
3) входные данные считываются и присваиваются переменным программы, в ходе выполнения которой значения этих переменных модифицируют-

ся, и полученные выходные данные записываются обратно на внешние носители в требуемом формате;

4) программа завершает работу.

Как видно из этой схемы, время жизни переменных в программе определяется временем выполнения самой программы; однако время жизни данных часто превышает время единственного выполнения программы, то есть они сохраняются и в периоды между ее запусками. В этом случае говорят, что данные *сохраняемы*.

Сегодня существует множество приложений, которые не вписываются в эту модель. Рассмотрим, например, электронную систему заказа авиабилетов. Для того чтобы заказать билет, вы связываетесь с агентом, и он делает запрос в системе регистрации предварительных заказов, запуская программы, которые сообщают сведения о расписании, наличии свободных мест и ценах. Данные и программы существуют в тесной взаимосвязи друг с другом. В этом случае было бы очень удобно воспользоваться языком, в котором предусмотрена возможность работы с сохраняемыми данными, что сделало бы подобные интерактивные системы более эффективными. В таком языке можно было бы объявлять переменные, чье время жизни не ограничивается временем выполнения программы. Создание программ упростилось бы за счет того, что не было бы необходимости указывать формат хранения данных во внешнем файле, перед тем как использовать их в программе. Транслятор языка имел бы информацию о местоположении и формате этих данных. В разделе 11.4.1 мы расскажем о современных исследованиях в области создания языков, позволяющих работать с сохраняемыми данными. Но поскольку эти языки не очень широко распространены, в настоящее время используются внешние файлы для передачи сохраняемых данных в локальные переменные программы (см. обсуждение в разделе 5.3.3).

5.1.2. Типы данных

Тип данных — это некоторый класс объектов данных вместе с набором операций для создания и работы с ними. Если программа имеет дело с какими-то конкретными *объектами данных* (например, массив `A`, целочисленная переменная `X` или файл `F`), язык программирования, как правило, по необходимости имеет дело с *типами данных* — классами массивов, целых чисел или файлов и операциями, позволяющими с ними работать.

В каждом языке имеется некоторый набор встроенных *примитивных* типов данных. Дополнительно в языке могут быть предусмотрены средства, позволяющие программисту определять новые типы данных. Одно из главных различий между ранними языками программирования, такими как FORTRAN или COBOL, и более поздними, такими как Java и Ada, лежит в области определяемых программистом типов данных (эта тема подробно рассмотрена в разделе 6.4). Современный подход к проблеме состоит в том, что в языке должны присутствовать средства для манипулирования типами данных. Это важнейшее свойство, добавленное в язык ML, и одна из особенностей моделей объектно-ориентированного программирования, обсуждаемых в главе 7.

Основные элементы *спецификации* типа данных следующие:

1) *атрибуты*, которые характеризуют объекты данных заданного типа;
2) *значения*, которые могут принимать объекты заданного типа;
3) *операции*, которые определяют возможные манипуляции над объектами данных заданного типа.

Например, в случае спецификации для типа данных «массив» атрибуты могут включать количество размерностей массива, допустимый диапазон изменения индекса для каждой размерности и тип данных элементов массива. Значения могут быть представлены множествами чисел, которые определяют допустимые значения элементов массива, а операции могут включать индексацию элементов массива (каждому элементу соответствует определенный индекс). Возможны также операции для создания массивов, изменения их формы, доступа к некоторым атрибутам (например, верхней и нижней границе диапазона изменения индексов) и выполнения арифметических действий над парами массивов.

Ниже перечислены основные элементы *реализации* типа данных.

1. *Способ представления* объектов данных этого типа в памяти компьютера в процессе выполнения программы.
2. *Способ представления операций*, определенных для этого типа данных, через конкретные алгоритмы и процедуры, которые используются для манипуляций с выбранной формой представления объектов данных в памяти. Реализация типа данных определяет, каким должно быть программное моделирование соответствующих частей виртуального компьютера в терминах более простых базовых конструкций, предоставляемых соответствующим слоем виртуального компьютера. Последний, в свою очередь, может быть представлен либо непосредственно аппаратными средствами, либо комбинацией аппаратных и программных средств, определенных операционной системой или микрокодом.

Последнее, что требуется определить при описании типа данных, — это его *синтаксическое представление*. И спецификация, и реализация мало зависят от конкретных синтаксических форм, используемых в данном языке. Атрибуты объектов данных обычно представлены синтаксически через *объявления* или *определения типов*. Значения могут представляться литералами или именованными константами. Вызов операций может происходить при помощи специальных символов, встроенных процедур или функций, таких как sin или read, или неявным образом через комбинации других элементов языка. Конкретный вид синтаксического представления не имеет большого значения, но синтаксическая информация может быть использована компилятором для определения времени связывания различных атрибутов и, следовательно, позволяет транслятору создать наиболее эффективное представление данных в памяти или выполнить проверку данных на соответствие указанному типу.

Спецификация элементарных типов данных

Элементарный объект данных может содержать только одно значение. Класс таких объектов, для которых определены различные операции, называется *элемен-*

тарным типом данных. Хотя в каждом языке программирования, как правило, присутствует свой, отличный от других набор элементарных типов данных и хотя конкретный вид их спецификации в различных языках может существенно различаться, тем не менее такие типы, как вещественные и целые числа, булевы значения, перечисления и указатели присутствуют почти во всех языках. Например, несмотря на то что булев тип данных включен в большинство языков, их спецификации в Java и C++ совершенно различны[1].

Атрибуты. Основные атрибуты любого объекта данных, в частности его имя и тип, остаются неизменными в течение времени жизни объекта. Во время работы программы некоторые атрибуты могут храниться в *дескрипторе* (описателе данных, который также называется *вектором предварительной информации*) как часть этого объекта данных; другие могут использоваться только для определения местоположения объекта в памяти компьютера и не использоваться явным образом. Обратите внимание на то, что *значение атрибута объекта* данных отличается от *значения, которое содержится в этом объекте.* Последнее может изменяться в течение времени жизни объекта и в процессе выполнения программы всегда представлено явным образом.

Значения. Тип объекта данных определяет множество допустимых значений, которые могут содержаться в этом объекте. Например, целочисленный тип определяет всевозможные целые числа, которые могут служить значениями объектов данных этого типа. В языке C определены четыре класса целых чисел: `int`, `short`, `long` и `char`. Поскольку в большинстве компьютеров арифметические операции на аппаратном уровне реализованы несколькими способами с различной точностью (например, 16-битные и 32-битные целые числа или 32-битные и 64-битные целые числа), в языке C программист может выбирать между этими различными аппаратными представлениями. Класс `short` соответствует наиболее короткому представлению числа, класс `long` — наиболее длинному аппаратному представлению, а класс `int` использует наиболее эффективный аппаратный способ представления из возможных. Этот класс может совпадать с классом `short` или `long`, а может соответствовать некоторому промежуточному представлению числа. Интересно отметить, что в C символы хранятся как 8-битные целые числа типа `char`, который является подтипом целочисленного типа.

Множество значений, допустимых для элементарных объектов данных, обычно является *упорядоченным множеством* с наименьшим и наибольшим элементами. В каждой паре различных значений (элементов упорядоченного множества) одно всегда больше другого. Например, для целочисленного типа данных имеются наибольшее значение, которое может быть представлено в памяти компьютера, наименьшее значение и все промежуточные целые числа в их обычной последовательности.

Операции. Набор операций, определенных для какого-то типа данных, задает возможные манипуляции (действия) с объектами этого типа. Эти операции могут быть *элементарными*, то есть являться частью определения языка, или они могут

[1] В C++ булевы значения true и false реализованы через целочисленный тип данных, и поэтому любое числовое значение (через приведение типов) может рассматриваться как булево, тогда как в Java булев тип данных — это самостоятельный тип, отличный от целочисленного. — *Примеч. науч. ред.*

определяться программистом в виде подпрограмм или методов как часть определения класса. В этой главе мы в основном будем заниматься элементарными операциями; определяемые программистом операции будут рассмотрены более подробно в следующих главах.

ПРИМЕР 5.3. Сигнатуры элементарных операций

1. Целочисленное сложение — это операция, аргументами которой являются два целочисленных объекта данных, а результатом — также целочисленный объект данных, обычно содержащий значение, равное сумме значений аргументов. Следовательно, спецификация выглядит следующим образом:

   + : integer × integer → integer

2. Операция «=» проверяет на равенство значения двух целочисленных объектов данных. Результатом выполнения этой операции является булево значение (истина или ложь). Спецификация может быть выражена так:

   = : integer × integer → Boolean

3. Операция извлечения квадратного корня, SQRT, применима к объектам данных вещественного типа:

   SQRT : real → real

Операция является математической функцией: каждому допустимому входному значению, то есть *аргументу*[1] (или набору аргументов), она сопоставляет однозначно определенный *результат*. У каждой операции имеются некоторая *область определения* (множество допустимых значений аргументов, на котором и определена эта операция) и *область значений* (множество возможных значений для результата выполнения операции). *Действие* операции определяет, каковы будут результаты для данного набора входных значений (аргументов).

Обычно действие операции задается с помощью *алгоритма*, который указывает, как следует вычислять результат для любого заданного набора аргументов; но возможны и другие варианты спецификации. Например, для того, чтобы определить действие операции умножения, вы можете просто привести таблицу умножения, в которой перечислены результаты перемножения любых пар чисел, вместо того чтобы указывать алгоритм перемножения двух чисел.

Для определения сигнатуры операции требуется указать количество и порядок следования аргументов из области определения операции, а также их тип; также нужно указать порядок следования и типы данных для области значения операции. Для такой спецификации удобно использовать обычный вид записи, используемый в математике:

имя операции : тип_арг × тип_арг × ... тип_арг → тип_результата

В языке С такая запись называется *прототипом* функции.

Операция, у которой имеются два аргумента и один результат, называется *бинарной*. Если у операции имеются один аргумент и один результат, то она называ-

[1] Аргументы операции называют также операндами. — *Примеч. науч. ред.*

ется *унарной*. Количество аргументов операции часто называется *арностью* операции. Большая часть примитивных операций в языках программирования бинарные или унарные.

Для точной спецификации действия операции обычно требуется информация более подробная, чем сигнатура. В частности, способ представления аргументов (точнее, типов данных, к которым относятся аргументы) в памяти компьютера обычно определяет, какие действия можно осуществлять с этими аргументами. Например, алгоритм перемножения двух чисел, представленных двоичным кодом, отличается от алгоритма перемножения десятичных чисел. Поэтому обычно в спецификации присутствует некоторое неформальное описание действия операции. Точная спецификация действия операции является частью реализации этой операции вместе с определением способа представления аргументов в памяти компьютера.

Иногда бывает непросто представить точную спецификацию операции в виде математической функции. Существуют четыре основных фактора, затрудняющих определение многих операций в языках программирования.

1. *Невозможность определить действие операции для некоторых аргументов.* Операция, определенная на некоторой области, может фактически быть не определенной для некоторых элементов этой области (например, операция извлечения квадратного корня не определена для отрицательных целых чисел). Иногда чрезвычайно трудно точно указать область, для которой не определена данная операции. В качестве примера можно привести множества чисел, которые вызывают переполнение или исчезновение значащих разрядов в арифметических операциях.
2. *Неявные аргументы.* Обычно при обращении к операции в программе ей передается некоторый набор явных аргументов. Однако в операции могут быть задействованы неявные аргументы — например, глобальные переменные или ссылки на другие нелокальные идентификаторы. Таким образом, полное выявление всех данных, которые влияют на результат операции, значительно усложняется наличием таких неявных аргументов.
3. *Побочные эффекты (неявные результаты).* Операция может не только выдавать явный результат (например, сумму чисел как результат операции сложения), но и модифицировать каким-то образом значения, хранящиеся в других объектах данных, как определяемых программистом, так и определяемых системой. Такие неявные результаты называются *побочными эффектами*. Функция может не только возвращать некоторое значение, но и изменять значения переданных ей параметров. Побочные эффекты являются основной частью многих операций, в частности таких, которые модифицируют структуры данных. Наличие побочных эффектов затрудняет точное определение области значения операции.
4. *Самоизменение (зависимость от предыстории).* Операция может изменять свою внутреннюю структуру — как локальные данные, которые сохраняются в промежутках между выполнением операции, так и свой собственный код. В таком случае для некоторого набора входных данных (аргументов) результат действия операции будет зависеть не только от этих аргументов,

но и от всей предыстории обращений к этой операции в течение работы программы и от переданных ей ранее аргументов. Такая операция называется *зависимой от предыстории*. Самый очевидный пример подобной операции — это *генератор случайных чисел*, который определен в качестве операции во многих языках. Обычно аргументом этой операции является некоторая константа, но результаты повторного выполнения операции различны. Действие операции заключается в том, что помимо генерации явного результата она также изменяет внутреннее *начальное число* (называемое также *затравкой*), которое влияет на результат последующего выполнения операции. Самоизменение посредством модификации локальных данных встречается достаточно часто; изменение собственного кода операции встречается реже, но все же возможно в таких языках, как LISP.

Подтипы. При описании нового типа данных часто возникает желание указать на его сходство с некоторым другим типом. Например, в языке C определены типы `int`, `short`, `long` и `char`, которые являются разновидностями целочисленного типа. Данные этих типов ведут себя сходным образом, и некоторые операции, в частности + и ×, было бы логично определить аналогичным образом для всех этих типов. Если некоторый тип данных является частью более крупного класса, то он называется *подтипом* этого класса, а сам класс называется *супертипом* для исходного типа данных. Например, в языке Pascal можно создать подкласс целых чисел следующим образом:

```
type SmallInteger = 1..20;
```

Этот подкласс состоит из целых чисел от 1 до 20 включительно.

Предполагается, что все операции, определенные для супертипа, допустимы также и для подтипа. Каким образом можно формализовать эту ситуацию? В языках Pascal и Ada поддиапазоны имеющихся в языке типов явным образом определены как части языка. Как можно расширить эти определения для других типов данных, которые не являются элементарными для языка? В главе 7 мы обсудим важную в этом отношении концепцию *наследования*, которая является обобщением указанного свойства подтипов. Можно сказать, что тип `char` в C наследует операции, определенные для целого типа `int`.

Реализация элементарных типов данных

Реализация элементарных типов данных складывается из *способа представления* объектов этого типа и их значений в памяти компьютера и набора *алгоритмов* и *процедур*, которые определяют операции с данными этого типа в терминах манипуляций, допускаемых избранным способом представления данных.

Способ представления

Способ представления элементарных типов данных сильно зависит от аппаратной части компьютера, на котором будет выполняться программа. Например, для целочисленных и вещественных величин почти всегда используется двоичное представление (соответственно целочисленное или с плавающей точкой), которое обеспечивается аппаратной частью компьютера. Причина такого выбора достаточно очевидна: при использовании того способа представления, который обеспечивает-

ся аппаратной частью компьютера, элементарные операции для этого типа данных можно реализовать при помощи операций, определенных с помощью аппаратной части; если же использовать какой-то иной способ представления данных, те же операции придется моделировать при помощи программ и они будут гораздо менее эффективными.

Атрибуты элементарных типов данных определяются аналогичным образом.

1. Для повышения эффективности во многих языках атрибуты данных определяются компилятором. Атрибуты данных не сохраняются непосредственно в представлениях данных во время выполнения программы. Такой метод характерен для языка C, в котором основными задачами являются эффективность использования памяти и скорость выполнения программы.
2. С другой стороны, атрибуты объекта данных могут в процессе выполнения программы содержаться в *дескрипторе* как часть самого объекта. Такой метод характерен для языков, в которых главной целью является гибкость, а не эффективность выполнения — например, в LISP и Prolog. Поскольку в большинстве случаев в аппаратной части компьютера непосредственно не предусмотрена возможность хранения дескриптора, то сами дескрипторы и операции над объектами, которые снабжены дескрипторами, должны быть программно-моделируемыми.

Способ представления объекта данных обычно не зависит от его местоположения в памяти. Он обычно описывается в терминах необходимого размера блока памяти (то есть количество необходимых слов, байтов или битов) и компоновки атрибутов и значений данных в пределах этого блока. Обычно адрес первого слова или байта в таком блоке памяти и представляет местоположение этого объекта данных.

Реализация операций. Любая операция, определенная для объекта данных заданного типа, может быть реализована одним из трех основных способов.

1. *Непосредственно в аппаратной части компьютера.* Например, если целые числа хранятся в памяти с использованием аппаратного представления, тогда сложение и вычитание целых чисел реализовано посредством арифметических операций, встроенных в аппаратную часть компьютера.
2. *Как процедура или подпрограмма-функция.* Например, операция извлечения квадратного корня, как правило, не представлена непосредственно в аппаратной части компьютера. Она может быть реализована как подпрограмма, вычисляющая квадратный корень некоторой величины, передаваемой ей в качестве параметра.
3. *Как встраиваемая последовательность кодов.* Встраиваемая последовательность кодов также является программно-моделируемой реализацией операции. Но в отличие от предыдущего варианта использования подпрограммы, в данном случае код этой подпрограммы копируется в выполняемую программу в том месте, где в противном случае пришлось бы вызывать подпрограмму. Например, функция, вычисляющая абсолютное значение (модуль) числа и определенная следующим образом:

   ```
   abs(x) = if x<0 then -x else x
   ```

   обычно реализуется как встраиваемая последовательность кодов:

- извлекается из памяти значение x;
- если x > 0, то следующий оператор опускается;
- выполняется присваивание x = -x;
- новое значение x сохраняется в памяти.

Каждая из этих строк реализована при помощи соответствующей операции, встроенной в аппаратную часть компьютера.

5.1.3. Объявления

При написании программы программист определяет имя и тип каждого объекта данных, который встречается в программе. Также должны быть определены время жизни каждого объекта данных, то есть часть программы, в которой этот объект используется, и те операции, которые применяются к этому объекту.

Объявление — это оператор программы, назначение которого заключается в том, чтобы сообщить транслятору информацию об именах и типах объектов данных, используемых в программе. Размещение объявлений в программе (например, внутри подпрограммы или определении класса) задает время жизни объектов. Например, в языке С объявление

```
float A, B;
```

в начале подпрограммы указывает, что во время выполнения этой подпрограммы потребуются два объекта данных типа float. Также это объявление связывает объекты данных с именами A и B на период времени их жизни.

Приведенное выше объявление языка С называется *явным объявлением*. Во многих языках используются также *неявные* объявления, или объявления *по умолчанию*. Они вступают в силу только в том случае, если отсутствуют явные объявления. Например, в подпрограмме на языке FORTRAN можно использовать простую переменную с именем INDEX, не объявляя ее явным образом, — по умолчанию компилятор предполагает, что эта переменная имеет целочисленный тип, так как ее имя начинается с буквы из диапазона I–N. В языке Perl объявление переменной происходит, когда ей присваивается некоторое значение:

```
$abc = 'a string'; # $abc - строковая переменная
$abc = 7;          #$ теперь abc - целочисленная переменная
```

В объявлении также можно задать значение объекта данных, если это константа, или начальное значение объекта данных, если он не является константой. В объявлении также могут быть указаны другие виды связываний для данного объекта: имя объекта или его размещение как компонента внутри другого, более крупного, объекта данных. Иногда в объявлении указываются и некоторые детали, относящиеся к реализации объекта: связывание с конкретным местоположением в памяти или с конкретной формой представления объекта. Например, используемый в языке COBOL модификатор COMPUTATIONAL для целых чисел обычно указывает на необходимость использования двоичного (а не символьного) представления значений этих объектов в памяти, что позволяет использовать более эффективные арифметические операции.

Объявление операций

Информация, которая в процессе трансляции требуется в первую очередь, — это сигнатура каждой операции. Обычно для встроенных в язык, то есть элементарных, операций явное объявление типов аргументов и результата не требуется. Эти операции можно вызывать по мере надобности, а типы аргументов и результатов для них по умолчанию определяются транслятором языка. Но для определяемых программистом операций обычно следует сообщить транслятору о типах аргументов и результатов перед тем, как обращаться к подпрограмме. Например, в языке C эта информация помещается в заголовочную часть определения подпрограммы, ее *прототип*. Так, следующий прототип

```
float Sub(int X, float Y)
```

объявляет подпрограмму Sub со следующей сигнатурой:

```
Sub: int × float → float
```

Функции объявлений

Объявления выполняют несколько важных функций.

1. *Выбор способа представления в памяти.* Если объявление содержит сведения о типе данных и атрибутах объекта данных, то на основе этой информации транслятор часто может оптимизировать представление этого объекта данных в памяти.
2. *Улучшение управления памятью.* Содержащаяся в объявлении информация о времени жизни объектов данных зачастую позволяет использовать более эффективные процедуры управления памятью во время выполнения программы. Например, в языке C все объекты, объявленные в начале подпрограммы, имеют одинаковое время жизни (равное времени выполнения подпрограммы) и, следовательно, могут быть расположены в едином блоке памяти, который создается при входе в подпрограмму и уничтожается при выходе из нее. Другие объекты данных в C создаются динамически посредством специальной функции malloc. Поскольку для таких объектов время жизни не указано, каждому такому объекту место в памяти отводится индивидуально.
3. *Полиморфные операции.* В большинстве языков используются специальные символы для обозначения некоторых основных операций; так, например, символ «+» может обозначать любую из нескольких операций сложения в зависимости от типа переданных аргументов. Например, в C запись A + B может обозначать «выполнить целочисленное сложение», если A и B принадлежат целочисленному типу, или «выполнить вещественное сложение», если A и B — вещественные числа. В этом случае говорят, что символ операции является *перегруженным*, так как он обозначает не одну конкретную операцию, а скорее некоторую *общую* операцию сложения (определяющую группу родственных операций), которая может иметь разные формы для аргументов разных типов. В большинстве языков основные символы операций, такие как +, * и /, являются перегруженными (то есть они обозначают общие операции); другие имена операций определяют операции однозначным образом. Но, например, в языке Ada программист имеет возможность опре-

делять перегружаемые имена подпрограмм и добавлять дополнительный смысл существующим символам операций. В языке ML концепция полиморфизма получила полную реализацию, в которой одно и то же имя функции может быть использовано для многих ее реализаций, зависящих от типа передаваемых аргументов. Если говорить об объектно-ориентированных языках, то полиморфизм является главным свойством, которое позволяет программисту расширить язык программирования с помощью новых типов данных и операций.

Объявления обычно позволяют компилятору языка определить во время компиляции, какая именно из группы родственных операций, обозначенных перегруженным символом, подразумевается в каждом конкретном случае. Например, в языке C на основе объявлений переменных `A` и `B` компилятор определяет, какая из двух возможных операций (целочисленное или вещественное сложение) подразумевается в записи `A + B`. В таком случае не требуется проверки соответствия типов при выполнении программы. В языке Smalltalk, напротив, требуется определять конкретный вид операции сложения каждый раз, когда при выполнении программы встречается символ «+», поскольку в этом языке отсутствуют объявления типов для переменных.

4. *Контроль типов.* С точки зрения программиста, наиболее ценным свойством объявлений является то, что они позволяют производить *статический*, а не *динамический* контроль типов.

5.1.4. Контроль типов и преобразование типов

Способы представления данных, поддерживаемых аппаратной частью компьютера, обычно не предусматривают включение информации о типе данных, а при выполнении элементарных операций над такими данными контроль типов не осуществляется. Например, некоторое слово в памяти компьютера во время выполнения программы может содержать последовательность бит 11100101100...0011. Эта последовательность может представлять собой целое число, вещественное число, цепочку литер или инструкцию процессора — не существует способа определить, что именно. Элементарная операция целочисленного сложения, встроенная в аппаратную часть компьютера, не может проверить, являются ли переданные ей два аргумента целыми числами, — для нее это просто последовательности битов. Таким образом, на уровне аппаратуры обычные компьютеры весьма ненадежны в плане обнаружения ошибок в типах данных.

Контроль типов означает проверку того, что каждая операция, выполняемая в программе, получает правильное количество аргументов правильного типа. Например, перед выполнением оператора присваивания

```
X = A + B * C
```

компилятор должен проверить, что каждая операция (умножение, сложение и присваивание) получает по два аргумента правильного типа. Если операция сложения + определена только для целых или вещественных чисел, а `A` принадлежит к символьному типу данных, то произойдет *ошибка в типе аргумента*. Контроль типов

может происходить во время выполнения программы (*динамический контроль типов*) или во время компиляции (*статический контроль типов*). Значительное преимущество использования в программировании языков высокого уровня объясняется тем, что в реализации таких языков предусмотрен контроль типов для всех (или почти всех) операций и таким образом программист оказывается защищен от этого весьма трудноуловимого вида ошибок.

Динамический контроль типов для аргументов некоторой операции осуществляется во время работы программы непосредственно перед выполнением данной операции. Реализация динамического контроля обычно основана на использовании дескриптора типа, который хранится вместе с объектом данных и указывает его тип. Например, в целочисленном объекте данных содержится и его значение, и дескриптор, указывающий на принадлежность объекта к целочисленному типу. Тогда каждая операция реализуется таким образом, чтобы в первую очередь проверялся дескриптор типа каждого аргумента. Операция выполняется, только если типы аргументов правильны; в противном случае генерируется сообщение об ошибке. После того как получен результат операции, к нему тоже должен быть присоединен соответствующий дескриптор, описывающий его тип, чтобы операции, в которых впоследствии этот объект может участвовать в качестве аргумента, могли бы проверить его тип.

Некоторые языки программирования, например Perl и Prolog, разработаны таким образом, что для них необходим динамический контроль типов. В этих языках не используется явное объявление типов переменных и отсутствуют встроенные (неявные) объявления типов (в отличие от языка FORTRAN, в котором переменные объявляются по умолчанию). Тип переменных `A` и `B` в выражении `A + B` может меняться в процессе выполнения программы. В такой ситуации динамический контроль типов этих переменных необходим для каждой операции сложения, происходящей во время выполнения программы. В таких языках, где отсутствуют объявления, переменные называются *не имеющими типа*, или *бестиповыми*, так как у них действительно нет никакого фиксированного типа.

Основным преимуществом динамического определения типов является гибкость программы: объявления типов отсутствуют, и тип объекта данных, связанного с некоторым именем, может меняться по мере необходимости в процессе выполнения программы. Программист, таким образом, лишается большей части забот, связанных с типами данных. Но, с другой стороны, динамический контроль типов имеет несколько существенных недостатков.

1. Затрудняется отладка программы (то есть становится трудно удалить все ошибки, связанные с типами аргументов). Поскольку динамический контроль происходит во время выполнения операции, для некоторых операций типы аргументов могут остаться непроверенными (такая ситуация возникает, если для конкретных входных данных эти операции относятся к невыполняемым ветвям программы). Вообще говоря, в процессе тестирования программы невозможно проверить все ее ветви. Все непроверенные ветви могут содержать ошибки, связанные с типами аргументов.

2. Динамический контроль типов подразумевает, что вся информация о типах должна сохранятся на всем протяжении работы программы, что требует дополнительных объемов памяти, иногда весьма существенных.

3. Динамический контроль типов должен быть, как правило, реализован посредством программного моделирования, поскольку аппаратная часть компьютера чаще всего не обеспечивает таких возможностей. Это приводит к замедлению выполнения программы.

В большинстве языков предприняты попытки свести на нет или минимизировать динамический контроль типов путем замены его на *статический контроль*, то есть контроль во время компиляции. Необходимая для статического контроля информация частично поступает из объявлений, которые явным образом создает программист, а частично — из других языковых структур. Необходимая для реализации статического контроля типов информация должна включать в себя следующее.

1. *Для каждой операции должны быть указаны количество, порядок и тип данных для аргументов и результатов* (иначе говоря, сигнатура операции).
2. *Для каждой переменной, представляющей собой имя объекта данных, следует указать тип этого объекта.* Тип объекта данных не должен изменяться в процессе выполнения программы (он должен быть инвариантом). Однако при проверке типов для выражения A + B можно предположить, что тип данных для объекта под именем A остается неизменным при каждом вычислении выражения, даже если оно вычисляется в цикле и на каждом его шаге переменная A связывается с некоторым новым объектом данных.
3. *Должны быть указаны типы всех констант.* Синтаксическая форма записи литерала обычно дает информацию о его типе (например, 2 — это целое число, а 2.3 — вещественное). Для проверки типа именованной константы следует использовать ее определение.

На начальном этапе трансляции программ компилятор (или другой транслятор) собирает информацию о типах объектов данных из объявлений, имеющихся в программе, и вносит ее в различные таблицы, главным образом в таблицу символов (см. главу 3), которая содержит сведения о типах переменных и операций. После того как собрана вся эта информация, происходит проверка всех вычисляемых в программе операций на правильность типов их аргументов. Заметим, что, как отмечалось ранее, для полиморфных операций любой из нескольких возможных типов аргументов будет воспринят как правильный. Если все аргументы для данной операции прошли проверку, то определяется тип результата и полученная информация сохраняется компилятором для проверки последующих операций. Обратите внимание на то, что в случае полиморфной операции опять-таки общее (групповое) имя может быть заменено именем, специфическим для конкретного типа данных передаваемых ей аргументов.

Поскольку статический контроль типов осуществляется для всех без исключения операций, имеющихся в программе, то проверяются все возможные ветви программы и в дальнейшем контроле не возникает необходимости. Таким образом, не требуется включать в объекты данных дескрипторы типов и производить динамический контроль типов. В результате получается значительный выигрыш в скорости выполнения программы и эффективности использования памяти.

Стремление реализовать статический контроль типов оказывает значительное влияние на многие аспекты языка: объявления, структуры управления данными,

организацию раздельной компиляции подпрограмм и т. д. Во многих языках не всегда можно осуществить статический контроль типов для некоторых конструкций в определенных случаях. Подобные проблемы со статическим контролем типов в некоторых языковых структурах можно решать двумя способами.

1. *Отказаться от статического контроля в пользу динамического.* Цена такого решения часто оказывается высокой в отношении используемой памяти, так как во время выполнения программы требуется хранить дескрипторы типов для всех объектов данных, хотя они и редко проверяются.
2. *Вообще отказаться от контроля типов для операций.* Не проверенные на соответствие типов аргументы операции могут вызвать серьезные и трудноуловимые ошибки в программах, но иногда приходится использовать этот вариант, если стоимость динамического контроля типов оказывается слишком высокой.

Сильная типизация. Если мы можем при помощи статического контроля обнаружить все без исключения ошибки определения типов в программе на данном языке, то такой язык называется *сильно типизированным.* Вообще говоря, строгий контроль типов является некоторой гарантией отсутствия соответствующих ошибок в программе. Мы называем функцию f с сигнатурой f : S → R *безопасной в отношении типа,* если при вычислении этой функции результат не может выйти за пределы множества R. В случае всех операций, безопасных в отношении типа, мы знаем еще до выполнения программы, что тип результата будет правильным и что динамический контроль типов не требуется. Очевидно, если каждая операция безопасна в отношении типа, то язык в целом является сильно типизированным.

Однако немногие языки соответствуют этому требованию. Например, в языке С, если X и Y принадлежат типу short (то есть короткие целые числа), то результат операций X + Y или X * Y может оказаться за пределами этого типа, вследствие чего возникнет ошибка типа. Хотя реализация в языке настоящего строгого контроля типов трудна, можно значительно приблизиться к ней, если ограничить преобразование одного типа в другой, о котором пойдет речь в следующем разделе.

Выведение типа. В языке ML предложен интересный подход к определению типов данных. Суть его в том, что в случае, если возможна однозначная интерпретация типа данных, его объявление не требуется. Язык реализован таким образом, что недостающая информация о типах данных может быть выведена из других имеющихся объявлений типов. В этом языке для объявления аргументов функций используется достаточно стандартный синтаксис, а именно

```
fun area(length:int, width:int):int = length * width;
```

В данном случае функция area возвращает целое число — величину площади прямоугольника, которая вычисляется как результат перемножения двух целочисленных аргументов, длины и ширины. В данном случае, если определен тип хотя бы одного из этих трех объектов данных: area, length или width, то можно вывести типы двух других. То есть, даже исключив описание типов каких-то двух параметров, мы все же получаем однозначную интерпретацию этой функции. Поскольку символ «*» может означать и целочисленное, и вещественное перемножение, каждая из приведенных далее форм записи в ML эквивалентна записи из предыдущего примера:

```
fun area(length, width):int = length * width;
fun area(length:int, width) = length * width;
fun area(length, width:int) = length * width;
```

Однако такая запись

```
fun area(length, width) = length * width;
```

уже ошибочна, так как в ней отсутствует однозначность определения типов данных. В такой записи все используемые объекты данных могут быть либо целыми, либо вещественными.

Приведение и преобразование типов

Если в процессе проверки типов выявляется несоответствие между фактическим и ожидаемым типами аргумента для операции, то может произойти одно из двух:

1) несоответствие типов будет воспринято как ошибка, о чем появится сообщение, и будут предприняты предусмотренные для такого случая действия;
2) произойдет *приведение* (или *неявное преобразование*) типа[1] фактически переданного аргумента к правильному, ожидаемому для данной операции:

   conversion_op : $type_1 \rightarrow type_2$

Иначе говоря, преобразование типа заключается в том, что вместо объекта данных одного типа создается соответствующий ему новый объект другого типа. В большинстве языков преобразование типов осуществляется одним из следующих двух способов.

1. Набором встроенных функций, которые программист может вызвать явным образом для преобразования типа какого-либо объекта. Например, в языке Pascal имеется функция round, которая преобразует объект данных вещественного типа в целочисленный объект, значение которого равно округленному значению вещественного числа. В языке С мы принудительно приводим (cast) выражение к правильному типу. Например, операция (int) X преобразует вещественное X в объект целого типа.
2. Приведением типа, которое осуществляется автоматически в определенных случаях несоответствия типа. Например, в языке Pascal, если арифметической операции сложения переданы аргументы различных типов — вещественного и целочисленного, то по умолчанию целочисленный тип преобразуется в вещественный и затем осуществляется вещественное сложение. В отличие от С++ в Java неявное приведение типов допускается, если сама операция допускает *расширение*. Таким образом, в Java целочисленное значение может быть присвоено переменной вещественного типа, а в С++ следует преобразовать тип значения в вещественный явным образом.

В основе неявного приведения типов лежит концепция недопущения потери информации. Имеется в виду следующее: поскольку в языке С любое короткое целочисленное значение может быть преобразовано в длинное, при автоматическом преобразовании short int → long int не происходит потери информации. Такое приведение называется *расширением*. Аналогично, поскольку в большин-

[1] *Преобразование типа* — это операция с сигнатурой.

стве языков целые числа могут быть однозначно выражены через вещественные объекты данных, короткие целые числа могут быть расширены до вещественных без какой-либо потери информации.

Но преобразование вещественного числа в целое, как правило, связано с потерей информации. Хотя 1,0 в точности равно 1, число 1,5 уже невозможно преобразовать в целое число. Оно будет преобразовано в 1 или 2. В данном случае приведение типа называется *сужением* и приводит к потере информации.

При динамическом контроле типов приведение осуществляется в момент обнаружения несоответствия типов при выполнении программы. В таких языках сужения могут быть допустимы для некоторых значений объектов данных (например, 1,0 можно преобразовать к целому типу, но 1,5 нельзя). При статическом контроле типов для реализации таких сужений в программу при компиляции в нужных точках вставляется дополнительный код, который во время выполнения программы и осуществляет сужение. Но так как эффективность работы программы обычно является одним из основных требований, программисты стараются избегать применения сужений, чтобы не приходилось выполнять дополнительный код, определяющий их допустимость.

Операции преобразования типов могут потребовать значительных затрат компьютерного времени при выполнении программы. Например, в языках COBOL и PL/I числа часто хранятся в виде символьных строк. В большинстве машин операция сложения реализована таким образом, что эти символьные строки приходится преобразовывать в двоичный код, поддерживаемый аппаратной частью компьютера, а результат сложения снова преобразовывать в символьные строки для хранения в памяти. Затраты времени на преобразование типов могут превосходить затраты на само сложение в сотни раз.

Разработчики трансляторов языков тем не менее иногда смешивают семантику объектов данных и способы их представления в памяти. Примером могут служить десятичные числа в COBOL и PL/I. Трансляторы PL/I обычно хранят данные типа FIXED DECIMAL в упакованном десятичном формате. Это представление обеспечивается непосредственно аппаратной частью компьютера, но все же выполняется достаточно медленно. В компиляторе PL/C [32] данные типа FIXED DECIMAL хранятся в виде 16-значных вещественных чисел двойной точности с плавающей точкой. При таком способе хранения не происходит потери точности (что особенно важно при хранении десятичных чисел). Например, вычисление суммы 123.45 + 543.21 приводит к довольно медленному сложению упакованных десятичных чисел (или еще более медленному программно-моделируемому сложению, если упакованные десятичные данные не поддерживаются непосредственно аппаратной частью компьютера). В компиляторе PL/C это же сложение осуществляется как единое действие, требующее гораздо меньших временных затрат: складываются два числа с плавающей точкой 112345 + 54321, а компилятор следит за положением десятичной точки (в данном случае результат сложения должен быть умножен на 10^{-2}), являющейся атрибутом результата, определяемым во время компиляции.

Существует два противоположных мнения относительно «широты» применения неявного преобразования типов. В языках Pascal и Ada оно почти не используется; любое несоответствие типов, за небольшим исключением, воспринимается

как ошибка. В С приведение типов является правилом — когда компилятор обнаруживает несоответствие типов, он пытается отыскать подходящую операцию преобразования, которую можно было бы вставить в компилируемый код для того, чтобы изменить нужным образом тип объекта данных. Сообщение об ошибке возникает только в том случае, когда компилятор не может произвести требуемое преобразование типов.

Несоответствие типов — это хотя и незначительная, но распространенная ошибка, и поэтому необходимость в преобразовании типов возникает достаточно часто, особенно в тех языках, в которых определено большое количество типов данных. Но значение понятия «несоответствие типов» не так очевидно и с ним связан ряд сложных вопросов (см. раздел 6.4). Приведение типов часто освобождает программиста от необходимости кропотливой работы, которая потребовалась бы для явного введения в программу операций преобразования типов. С другой стороны, приведение типов может скрыть наличие других, более серьезных ошибок, которые в противном случае могли бы быть замечены программистом при компиляции.

Язык PL/I, в частности, печально известен как язык, компиляторы которого имеют тенденцию так «исправлять» незначительные ошибки (например, неправильно написанное имя переменной) посредством сложных преобразований типа, что их становится очень трудно обнаружить. Поскольку в PL/I допускаются сужающие преобразования типов, иногда получаются удивительные результаты. Например, `9 + 10/3` является недопустимым выражением! Чтобы понять это, заметим, что `10/3` преобразуется к виду `3.3333333...` (цифра 3 повторяется столько раз, сколько допускается реализацией языка на данном компьютере). Но в результате операции сложения `9 + 3.33333...` получается ошибка переполнения, так как в целой части результата добавляется еще одна цифра. Если эту ошибку проигнорировать, то результат автоматически преобразуется к виду `2.33333...`, а это совсем не то, что ожидалось.

5.1.5. Присваивание и инициализация

Большая часть операций для обычных элементарных типов данных — в частности, для чисел, перечислений, булевых величин и символов — выполняется следующим образом: берутся один или два аргумента одного типа, производятся какие-то относительно простые арифметические операции, операции вычисления отношений или другие и вычисляется результирующий объект данных, который может быть того же типа, что и аргументы, а может и отличаться от них. Операция присваивания выполняется более сложным образом и заслуживает отдельного рассмотрения.

Присваивание — это элементарная операция, которая изменяет связывание объекта данных со значением. Однако это изменение — *побочный эффект* данной операции. В некоторых языках, например в С и LISP, операция присваивания также возвращает некоторый результат — объект данных, содержащий копию присвоенного значения. Эти факторы становятся более понятными, когда мы пытаемся написать спецификацию для операции присваивания. В Pascal спецификация для операции присваивания выглядит так:

присваивание(:=) : $целое_1 \times целое_2 \rightarrow$ пустой тип (void)

Действие операции таково: установить значение, содержащееся в объекте данных $целое_1$, таким образом, чтобы оно стало копией значения, содержащегося в объекте данных $целое_2$, но не возвращать явный результат. (Изменение значения объекта $целое_1$ является неявным результатом или побочным эффектом этой операции.) В С спецификация операции присваивания выглядит иначе:

присваивание(=): $целое_1 \times целое_2 \to целое_3$

Действие операции таково: установить значение, содержащееся в объекте данных $integer_1$, таким образом, чтобы оно стало копией значения, содержащегося в объекте данных $integer_2$, а также создать и возвратить новый объект данных $integer_3$, содержащий копию значения объекта $integer_2$.

Рассмотрим такое присваивание: X := X. Здесь интересно наблюдать за различными интерпретациями ссылки на переменную X. Стоящее справа X обозначает величину, содержащуюся в переменной с этим именем. Подобные ссылки называются *правосторонними значениями* (оператора присваивания), или *r*-значениями, объекта данных. Аналогично стоящее слева X обозначает местоположение объекта, в котором будет содержаться новое значение. Такие ссылки называются *левосторонними значениями* (оператора присваивания), или *l*-значениями. Тогда мы можем определить оператор присваивания следующим образом:

1) сначала вычисляется *l*-значение первого операнда выражения;
2) затем вычисляется *r*-значение второго операнда выражения;
3) присваивается вычисленное *r*-значение вычисленному *l*-значению объекта данных;
4) возвращается вычисленное *r*-значение как результат выполнения операции.

Если у нас имеется операция присваивания (такая как, например, в С), то можно сказать, что она возвращает *r*-значение объекта данных, которому присвоили новое значение. Помимо того, в С имеется рад унарных операций для манипуляций с *r*-значениями и *l*-значениями выражений, которые позволяют выполнять множество полезных и порой странных действий, связанных с присваиванием. Такая двойственная «природа» операции присваивания — как функции изменения значения объекта данных (через его *l*-значение) и как функции, возвращающей значение (через *r*-значение объекта данных), — широко используется в С (более подробно см. главу 8).

Использование *r*-значений и *l*-значений позволяет более лаконично описывать семантику выражений. Рассмотрим операцию присваивания A = B, где A и B — целые числа. В языке С, как и во многих других языках, это означает «присвоить копию значения переменной B переменной A» (то есть «присвоить *l*-значению объекта A *r*-значение объекта B»). Рассмотрим теперь операцию присваивания A = B, где A и B — переменные-указатели. Если B — это указатель, то *r*-значение B является *l*-значением некоторого другого объекта данных. Тогда это присваивание означает следующее: «сделать *r*-значение объекта A ссылкой на тот же самый объект, указателем на который является *r*-значение объекта B». Иначе говоря, «присвоить *l*-значению объекта A *r*-значение объекта B, которое является *l*-значением некоторого другого объекта данных». Таким образом, присваивание A = B

означает «присвоить копию указателя, хранящегося в переменной B, переменной A» (рис. 5.2).

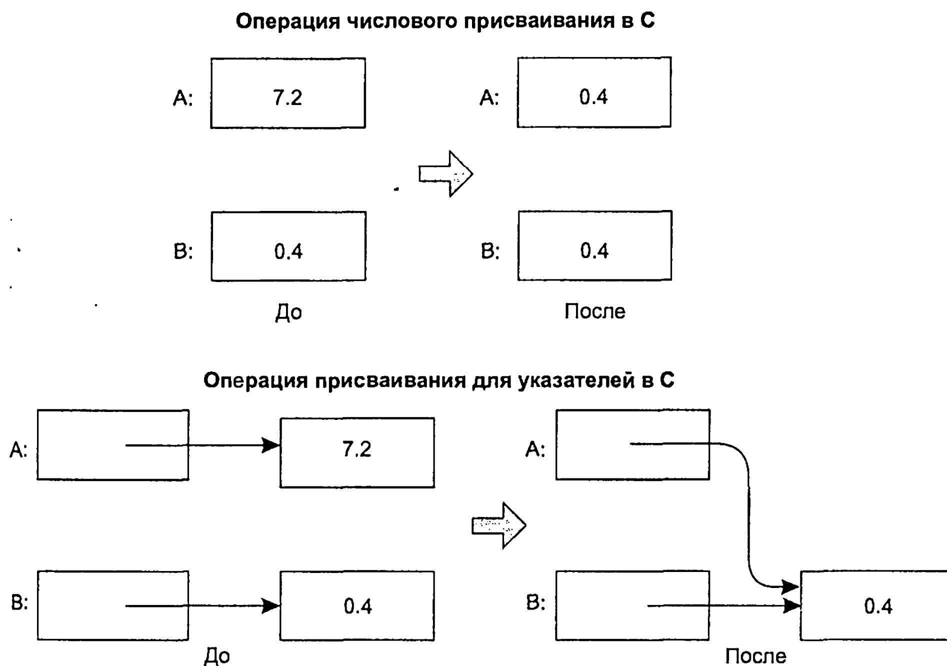


Рис. 5.2. Два взгляда на операцию присваивания

Равенство и эквивалентность. Оператор присваивания настолько распространен, что немногие задумываются о его семантике. Тем не менее существует один вопрос, требующий разрешения. Рассмотрим присваивание значения объекту A в некотором новом языке Zork:

```
A ← 2 + 3
```

Эту запись можно интерпретировать двумя способами:

1) вычисляется выражение 2 + 3 и его результат, то есть 5, присваивается A;

2) A присваивается целая операция 2 + 3.

В тех языках, где данные имеют статически определяемый тип, способ интерпретации зависит от типа объекта A. Если объект A определен как целое число, то имеет смысл только первая интерпретация; если же объект A определен как объект типа *операция*, то следует выбрать вторую интерпретацию. Но в языках с динамическим определением типов, когда тип объекта A задается посредством присвоения этому объекту какого-либо значения, допустимы обе интерпретации, и, следовательно, приведенная выше операция присваивания становится неоднозначной.

Именно такая ситуация возникает в языке Prolog. Операция is означает присваивание эквивалентного значения, в то время как операция = обозначает присваивание образца. Тогда равенство определяется в зависимости от текущего зна-

чения, присвоенного переменной. Рассмотрим следующие две последовательности операций языка Prolog:

```
X is 2 + 3. X = 5
X = 2 + 3. X = 5
```

Первая последовательность правильная, так как сначала переменной X присваивается значение 5 и ее тип устанавливается как целочисленный, в то время как во второй последовательности переменной X сначала присваивается операция 2 + 3 — отнюдь не то же самое, что целое число 5.

Заметим, что в данном случае символ = используется в зависимости от контекста как для обозначения операции присваивания, так и для обозначения операции логического сравнения. Фактически здесь приведен пример использования принципа унификации, что характерно для языка Prolog (присваивание и 2 + 3, и 5 объекту X во второй последовательности операций взаимно несовместимо). В дальнейшем мы поясним это в разделе 8.4.3.

Инициализация. *Неинициализированная переменная*, или, более обобщенно, *неинициализированный объект данных*, — это такой объект, который был создан, но которому еще не было присвоено значение (иначе говоря, *l*-значение без соответствующего *r*-значения). Создание объекта данных обычно только подразумевает выделение блока памяти для этого объекта. Если далее не предпринимается никаких действий, то в этом блоке памяти остается та же комбинация битов, которая была там на момент создания объекта. Обычно для связывания объекта данных с каким-либо допустимым значением требуется явная операция присваивания. В некоторых языках (например, в Pascal) инициализация должна производиться явным образом, посредством операций присваивания. В других языках (например, в APL) при создании объекта данных необходимо сразу же задать его исходное значение, причем это происходит неявным образом, без использования программистом оператора присваивания.

Неинициализированные переменные являются серьезным источником ошибок как для новичков, так и для профессионалов. Случайная конфигурация битов, содержащаяся в области, отведенной под значение неинициализированного объекта данных, по существу, неотличима от настоящего значения, которое также представляет собой некоторую конфигурацию битов. Таким образом, программа может использовать в вычислениях такое случайное «значение» неинициализированной переменной, и хотя ее работа может выглядеть правильной, по существу, в ней будет содержаться серьезная ошибка. Такое влияние на надежность программы неинициализированных переменных привело к тому, что, как правило, переменные принято инициализировать непосредственно при их создании путем присвоения им начальных значений, и в современных языках, например в Ada, предусмотрены механизмы, которые облегчают эту задачу. В Ada каждое объявление переменных может включать в себя также присваивание этой переменной начального значения, при этом используется синтаксис обычной операции присваивания. В частности, объявление

```
A: array(1..3) of float := (17.2, 20.4, 23.6);
```

создает массив A и явным образом присваивает каждому элементу массива начальное значение. Поскольку обычно массив создается динамически в процессе выполнения программы, реализация присваивания начальных значений требует генерации компилятором кода, который при выполнении программы явным образом присваивает указанные начальные значения объекту данных.

5.2. Скалярные типы данных

Сначала мы рассмотрим класс элементарных объектов данных, называемый *скалярными объектами данных*. Этот тип характеризуется тем, что у каждого объекта имеется только один атрибут. Например, для целочисленного объекта единственным атрибутом является его значение (например, 1, 3, 42) и никакой другой информации в этом объекте не содержится. Далее, в разделе 5.3, мы рассмотрим более сложные, составные объекты данных, у которых может иметься несколько атрибутов. Например, строка символов содержит последовательность символов (значение объекта), а дополнительным атрибутом может являться ее длина, то есть количество символов в строке.

Вообще, скалярные объекты данных (целых чисел, чисел с плавающей точкой, символов) тесно связаны с архитектурой аппаратной части компьютера, а составные объекты (например, строки) обычно представляют собой более сложные структуры, создаваемые компилятором, и не являются элементарными объектами, реализуемыми в аппаратной части компьютера.

5.2.1. Численные типы данных

Почти в любом языке программирования присутствуют в той или иной форме цифровые (численные) типы данных. Наиболее распространенными являются целочисленный и вещественный типы, так как они обычно поддерживаются аппаратной частью компьютера. Компьютерное представление числовых данных и реализация арифметических операций над ними сильно отличаются от чисел и операций, используемых в обычной математике. Компьютерная арифметика — это отдельный и очень увлекательный предмет, которому посвящены многие книги; но он не является центральным для понимания общей структуры языков программирования, поэтому здесь мы приводим только краткий обзор по данной теме.

Целые числа

Спецификация. Множество чисел этого типа образует ограниченное упорядоченное подмножество бесконечного множества целых чисел, определенного в математике. Максимальное значение иногда определяется как именованная константа (например, в Pascal имеется константа maxint). В таком случае обычно все множество допустимых значений целых чисел заключено в диапазоне между числами maxint и maxint. Как было сказано ранее, в языке С имеются четыре подтипа целочисленных объектов: int, short, long и char.

Операции над целочисленными объектами данных обычно разделяются на следующие основные группы.

- *Арифметические операции.* Сигнатура бинарных арифметических операций такова:

  БинОп : целое × целое → целое

  Здесь БинОп может быть сложением (+), вычитанием (−), умножением (×), делением (/ или div), а также делением по модулю (mod) или схожей операцией. Унарные операции имеют следующую сигнатуру:

  УнарОп : целое → целое

где УнарОп может быть, например, операцией отрицания (−) или идентичности (+). Обычно включаются и другие арифметические операции, часто в виде библиотечных подпрограмм-функций (например, операция определения *абсолютного значения*).

- *Операции сравнения.* Сигнатура любой операции сравнения выглядит следующим образом:

  СравОп : целое × целое → булево

  Здесь под СравОп могут подразумеваться следующие операции: равно, не равно, больше чем, меньше чем, больше или равно, меньше или равно. Операция сравнения сравнивает значения двух переданных ей аргументов и в качестве результата возвращает логический (булев) объект данных.

- *Операции присваивания.* Присваивание для целочисленных объектов можно определить одним из двух способов:

  присваивание : целое × целое → пустой тип

  или

  присваивание : целое × целое → целое

  Этот вопрос уже обсуждался в предыдущем разделе.

- *Битовые операции.* В языках с небольшим количеством элементарных типов данных целые числа выполняют множество разнообразных функций. В C, например, они используются в качестве булевых величин. Поэтому битовые операции также определены через целые числа:

  БитОп : целое × целое → целое

  В C имеются операции для побитовых логических и (&), или (|), сдвига битов (<<) и некоторые другие.

Реализация. Чаще всего определенный в языке целочисленный тип данных реализуется при помощи аппаратного представления целых чисел и набора элементарных арифметических операций и операций сравнения, реализованных аппаратурой компьютера.

На рис. 5.3 представлены три возможных способа представления целых чисел. В первом случае дескриптор типа отсутствует, в объекте содержится только само значение числа. Это представление допустимо, если в языке предусмотрены объявления и статический контроль типов для целочисленных объектов данных. Во втором случае дескриптор хранится в специально выделенной области памяти вместе с указателем местоположения фактического значения. Такой способ используется, в частности, в языке LISP. Недостаток этого представления заключается в том, что требуемый для хранения одного целого числа объем памяти может увеличиться вдвое; к достоинствам можно отнести использование для хранения числа встроенного аппаратного представления, что позволяет применять встроенные в аппаратную часть компьютера арифметические операции. Третий способ заключается в том, чтобы хранить число и дескриптор вместе, в одной ячейке памяти. Для этого приходится уменьшить размер целого числа (то есть количество бит в его представлении) настолько, чтобы освободилось место для дескриптора. В таком варианте получается выигрыш в отношении требуемого объема памяти, зато при использовании встроенных в аппаратуру арифметических операций требуется

сначала удалить дескриптор, затем выполнить операцию и после этого снова вставить дескриптор. Таким образом, для выполнения одной-единственной арифметической операции требуется осуществить целый ряд аппаратных действий, что приводит к снижению эффективности. Такой способ удобен только для реализованных в аппаратной части дескрипторов, что в настоящее время встречается довольно редко.

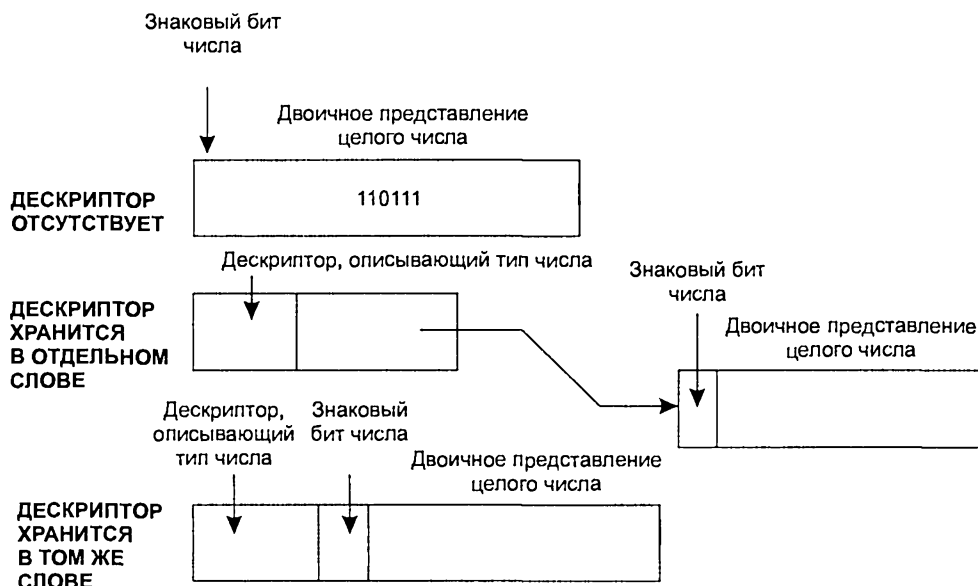


Рис. 5.3. Три способа представления в памяти целых чисел

Поддиапазоны

Спецификация. *Поддиапазон* целых чисел представляет собой подтип целого типа и содержит последовательность целых чисел из некоторого ограниченного их диапазона (например, это могут быть числа от 1 до 10 или от –5 до 50). Для определения подтипа часто используется объявление вида `A : 1..10` (Pascal) или `A : integer range 1..10` (Ada). Для элементов типа поддиапазон определены те же операции, что и для всего целочисленного типа; таким образом, такой подтип является частью базового *целочисленного типа*.

Реализация. Поддиапазоны оказывают двоякое влияние на реализацию.

1. *Уменьшаются затраты памяти.* Так как при использовании поддиапазона необходимо представлять только часть возможных значений целых чисел, то для хранения элементов этого подтипа требуется меньший объем памяти, чем для обычных целых чисел, то есть элементы этого типа могут быть представлены меньшим количеством бит. Например, для представления целых чисел из поддиапазона `1..10` требуется только четыре бита, в то время как для полного представления целых чисел на стандартном компьютере может потребоваться 16, 32 или более бит. Тем не менее, поскольку арифметические операции с укороченными целыми числами могут по-

требовать программного моделирования, при реализации таких чисел обычно используются минимальные по размеру последовательности бит, для которых в аппаратной части компьютера предусмотрены арифметические операции. Обычно это 8- или 16-битовые последовательности. Например, в C символы хранятся как 8-битовые целые числа, действия над которыми можно выполнить непосредственно при помощи команд большинства микропроцессоров.

2. *Улучшается контроль типов.* Если переменная объявлена как относящаяся к некоторому поддиапазону, то проверка значений, которые присваиваются этой переменной, становится более точной. Например, если переменная `Month` объявлена как `Month: 1..12`, то операция присваивания

   ```
   Month := 0
   ```

   ошибочна, что обнаруживается во время компиляции. Если же переменная `Month` объявлена просто как целочисленная, то приведенная операция присваивания формально оказывается правильной и программисту придется отыскивать эту ошибку уже при тестировании программы. Тем не менее далеко не всегда проверка подтипа может осуществляться во время компиляции. Если проверка затрагивает какое-либо вычисляемое значение, например

   ```
   Month := Month + 1
   ```

   она произойдет только при выполнении программы, когда станет понятно, не выходит ли полученное значение переменной `Month` за пределы значений поддиапазона. В таком случае требуется, чтобы во время выполнения программы были известны границы поддиапазона (например, `Month = 12` допустимо, а `Month = 13` недопустимо).

Вещественные числа с плавающей точкой

Спецификация. Вещественные числа с плавающей точкой обычно определяются одним атрибутом, например атрибутом `real` в языке FORTRAN или `float` в языке C. Как и в случае целочисленного типа, все допустимые значения вещественного типа формируют упорядоченную последовательность вещественных чисел из диапазона, обусловленного возможностями аппаратной части компьютера от минимального отрицательного значения до максимального положительного, но в отличие от целых чисел вещественные числа распределены в этом диапазоне неравномерно. В некоторых языках, например в Ada, требуемая для представления вещественных чисел точность может быть указана программистом в терминах количества значащих цифр в десятичном представлении.

Для вещественных чисел обычно предусмотрены те же операции (присваивание, арифметические операции и операции сравнения), что и для целых, хотя иногда набор операций отношения может быть ограничен. Из-за проблем, связанных с округлением, точного равенства двух вещественных чисел редко удается достичь. Если в программе имеется цикл, выход из которого обусловлен равенством вещественных чисел, такая программа может никогда не завершиться. По этой причине разработчик языка может, в принципе, запретить операцию проверки равенства двух вещественных чисел. Помимо перечисленных опера-

ций в большинстве языков определены в виде встроенных функций такие операции, как sin (синус) или max (максимальное из двух значений вещественных чисел):

```
sin : вещественное → вещественное
```

и

```
max : вещественное × вещественное → вещественное
```

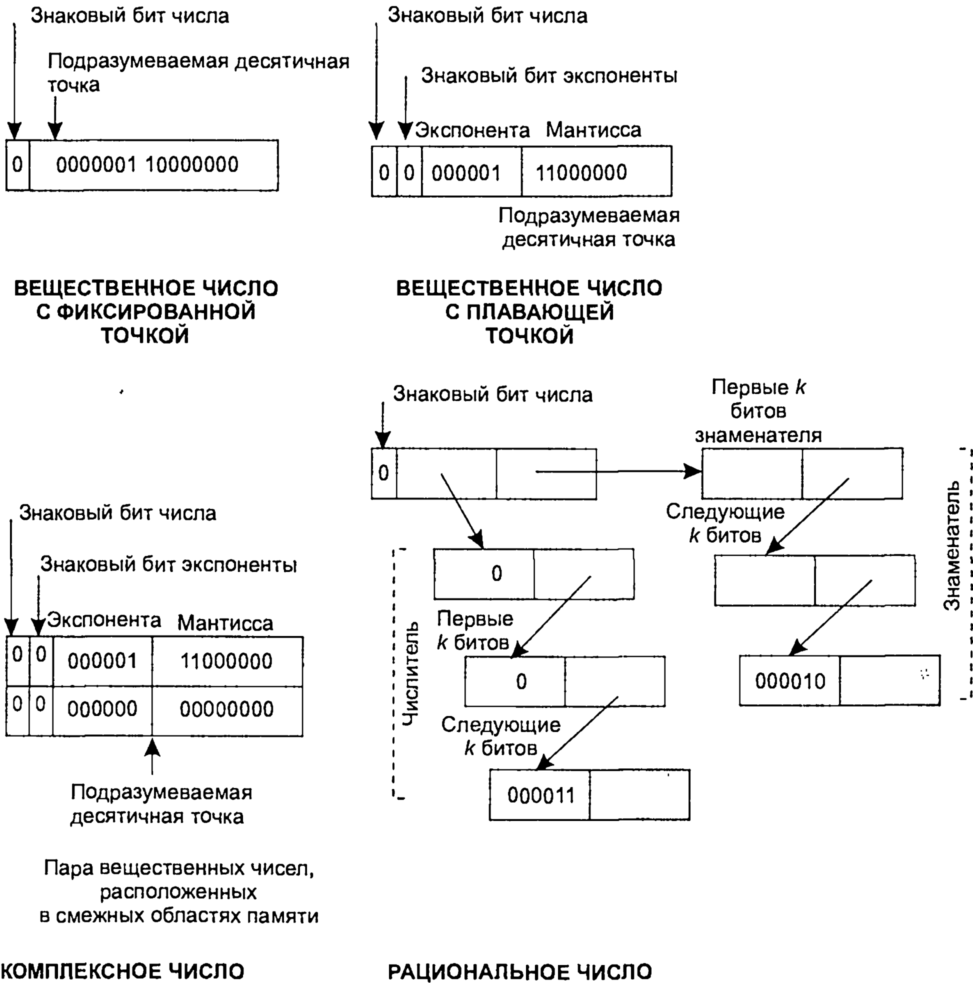


Рис. 5.4. Представление вещественного числа 1,5 без дескрипторов

Реализация. Хранение в памяти компьютера вещественных чисел с плавающей точкой обычно основано на аппаратном представлении, причем отводимая для хранения вещественных чисел область памяти разбивается на две части (см. рис. 5.4): мантиссу (в ней хранятся значащие цифры) и экспоненту (в ней хранится показатель степени). Такой способ хранения эмулирует математическую запись, в которой любое вещественное число может быть представлено как $N = m \times 2^k$, где

m — некоторое число из промежутка от 0 до 1, a k — целое число. Общепринятым определением формата вещественных чисел с плавающей точкой стал стандарт IEEE Standard 7754 [55] (см. пример 5.4).

Пример 5.4. IEEE формат чисел с плавающей точкой

Стандарт IEEE 754 определяет и 32-битовое, и 64-битовое представления чисел с плавающей точкой. 32-битовое представление выглядит слѐдующим образом:

Знак (S)	Экспонента (E)	Мантисса (M)
1	8	23

Числа представлены с помощью трех полей:

- S — однобитовое поле, хранящее знак числа. Значение 0 соответствует положительным числам;
- E — экспонента со сдвигом на величину 127. Под это поле отведено 8 бит, таким образом, имеется 256 различных значений от 0 до 255, соответствующих степеням 2 в диапазоне от −127 до 128 (с учетом сдвига);
- M — мантисса, под которую отведено 23 бит. Так как в нормализованной форме представления вещественных чисел первый бит в мантиссе всегда 1, то его можно опустить и вставлять автоматически аппаратными средствами, тем самым увеличивая точность до 24 бит.

S определяет знак числа. При заданных E и M значение числа получается следующим образом:

Параметры	Значение
E = 255 и M ≠ 0	Недопустимое значение
E = 255 и M = 0	∞
0 < E < 255	$2^{E-127}(1.M)$
E = 0 и M ≠ 0	$2^{-126}.M$
E = 0 и M = 0	0

Приведем несколько примеров:

$+1 = 2^0 \times 1 = 2^{127-27} \times (1).0$(двоичное) = 0 01111111 000000...
$+1.5 = 2^0 \times 1.5 = 2^{127-127} \times (1).1$(двоичное) = 0 01111111 100000...
$-5 = -2^2 \times 1.25 = 2^{129-127} \times (1).01$(двоичное) = 1 10000001 010000...

Таким образом, диапазон значений получается от 10^{-38} до 10^{+38}. В 64-битовом представлении поле для экспоненты расширяется до 11 бит, расширяя диапазон ее изменения от −1022 до +1023, что приводит к представлению десятичных чисел из диапазона от 10^{-308} до 10^{+308}.

Также используется представление вещественных чисел с двойной точностью, то есть с добавлением дополнительного слова к отводимой области памяти для хранения расширенной мантиссы. Обычно встроенные в аппаратную часть арифметические операции (сложение, вычитание, умножение и деление) могут выполняться для чисел и с одинарной, и с удвоенной точностью. Возведение

в степень обычно реализуется при помощи программного моделирования. Если поддерживаются числа и с одинарной, и с двойной точностью, то при определении того, какое представление требуется для конкретного объекта данных, используется обычное объявление объекта вещественного типа с указанием количества требуемых значащих цифр в представлении его значений. С другой стороны, программист может просто объявить вещественную переменную с помощью специальных ключевых слов, например double или long real, чтобы сообщить компилятору, что эта переменная требует представления с двойной точностью.

Вещественные числа с фиксированной точкой

Спецификация. Хотя в большинстве компьютеров аппаратная часть обеспечивает представление целых чисел и чисел с плавающей точкой, во многих приложениях требуется специальный вид рациональных чисел. Например, объекты данных, представляющие денежные суммы, должны содержать доллары и центы, то есть выглядеть как десятичные рациональные числа с двумя знаками после десятичной точки. С помощью целых чисел такая запись невозможна, а использование чисел с плавающей точкой может привести к ошибкам округления. Для подобных целей можно использовать представление вещественных чисел с фиксированной точкой.

Число с фиксированной точкой представляется как последовательность цифр фиксированной длины с десятичной точкой, расположенной в определенном месте между двумя цифрами. Объявление чисел с фиксированной точкой, например, в COBOL осуществляется с помощью ключевого слова PICTURE и выглядит следующим образом:

```
X PICTURE 999V99.
```

Здесь X объявляется как переменная с фиксированной точкой, тремя цифрами перед ней и двумя — после.

Реализация. Тип вещественных чисел с фиксированной точкой может поддерживаться непосредственно аппаратной частью компьютера или, как было сказано ранее, смоделирован программным способом. Например, в языке PL/I переменные этого типа объявляются как FIXED DECIMAL. Можно записать:

```
DECLARE X FIXED DECIMAL (10,3).
        Y FIXED DECIMAL (10,2).
        Z FIXED DECIMAL (10,2);
```

Это означает, что переменная X может содержать вещественные числа с тремя цифрами после десятичной точки, с общим числом используемых в представлении числа цифр 10; для представления чисел, хранящихся в переменных Y и Z, также используется всего 10 цифр, но они имеют два дробных десятичных разряда. Такие данные хранятся как целочисленные, а десятичная точка (ее положение) является атрибутом объекта данных. Если X = 103.421, то r-значение X будет 103 421, а у объекта данных X будет атрибут под названием *масштабный коэффициент* (scale factor, SF), в данном случае равный трем — это будет служить указанием, что десятичная точка располагается перед третьей справа цифрой. Таким образом, в выражении

$$значение(X) = r\text{-}значение(X) \times 10^{-SF}$$

SF всегда будет равно трем независимо от *r*-значения X. Аналогично если значение Y равно 102.34, то это число будет храниться как целочисленное значение 10 234 с масштабным коэффициентом SF = 2.

Рассмотрим, как будет выполняться оператор

```
Z = X + Y
```

Если вы возьметесь за решение этой задачи с помощью бумаги и карандаша, в первую очередь вам придется выровнять положение десятичной точки. Поскольку в X число имеет три знака после десятичной точки, а в Y — только два, вам придется сдвинуть значение Y влево на одну позицию и полученная сумма будет также содержать три знака после десятичной точки, то есть получится, что SF = 3:

```
X = 103.421
Y = 102.34x    ← сдвиг на 1 позицию влево
Sum =  205.761масштабный коэффициент суммы равен 3
```

Сдвиг влево эквивалентен умножению *r*-значения переменной Y на 10. Фактически выполняемый код для вычисления X + Y, таким образом, представляется в виде X + Y × 10, причем SF=3. Поскольку у чисел переменной Z имеются только два знака после запятой (SF=2), то нам придется удалить один знак, то есть фактически поделить *r*-значение Z на 10. Тогда получится следующий код для вычисления значения переменной Z:

```
Z = (X + Y × 10) / 10
```

Если масштабный коэффициент известен во время компиляции (так всегда и бывает), транслятор знает, как масштабировать результаты. Аналогично при перемножении чисел с фиксированной точкой X × Y мы получаем результат, перемножая *r*-значения этих переменных и складывая затем масштабные коэффициенты:

```
Product = rvalue(X) × rvalue(Y)
SF = SF(X) + SF(Y)
```

Вычитание и деление осуществляются таким же образом.

Другие числовые типы данных

Комплексные числа. Комплексное число состоит из пары чисел, представляющих его вещественную и мнимую части. Объект данных этого типа легко может быть представлен при помощи блока из двух областей памяти, содержащих пару вещественных чисел. Операции над комплексными числами моделируются при помощи программ, так как маловероятно, чтобы они поддерживались аппаратурой. Например, сложение реализовать довольно просто — следует сложить по отдельности мнимые и вещественные части аргументов, но уже умножение является более сложной операцией, в которой взаимодействуют все четыре компонента аргументов.

Рациональные числа. Рациональное число является результатом деления двух целых чисел. Обычно рациональные числа включаются в языки программирования, чтобы избежать проблем с округлением и усечением, которые встречаются при использовании вещественных чисел с фиксированной и плавающей точкой. Желательно представлять рациональные числа в виде упорядоченной пары целых чисел неограниченной длины. Такие длинные целые числа обычно представляются в виде связанных цепочек областей памяти. На рис. 5.4 схематически изображены некоторые способы представления чисел различных типов.

5.2.2. Перечисления

Часто возникает потребность задать некоторое ограниченное множество допустимых символических значений для какой-либо переменной. Например, переменная StudentClass может принимать только четыре различных значения: Freshman, Sophomore, Junior и Senior, — которые соответствуют I, II, III и IV курсам. Аналогично переменная EmployeeSex (пол работника) может иметь только два значения: male (мужчина) и female (женщина). В более старых языках, таких как COBOL или FORTRAN, такие переменные объявлялись как целочисленные и их значениями становились некоторые произвольно выбранные целые числа (например, Freshman = 1, Sophomore = 2 и т. д. или male = 0, female = 1). Затем в программе эти объекты воспринимались и обрабатывались как обычные целые числа. Использование поддиапазонов в таких случаях могло уменьшить объем требуемой памяти, но все же программисту приходилось следить, чтобы к таким переменным не применялась никакая арифметическая операция, в данном случае лишенная смысла. Дело в том, что, хотя присваивание переменной EmployeeSex значения 2 или умножение StudentClass на female не имеет смысла для указанной интерпретации этих переменных, транслятор не в состоянии обнаружить такую ошибку.

В языках C, Pascal и Ada имеется специальный тип *перечисление*, который в большей степени позволяет программисту непосредственно определять подобные переменные и манипулировать ими.

Спецификация. Перечисление представляет собой упорядоченную последовательность различных элементов. Программист определяет как буквальные имена этих элементов, так и порядок их следования в перечислении непосредственно в объявлении переменной. Например, в C это объявление выглядит следующим образом:

```
enum StudentClass {Fresh, Soph, Junior, Senior};
enum EmployeeSex {Male, Female};
```

Поскольку в программе обычно используется много переменных одного и того же перечисляемого типа, то довольно часто перечисление определяется как отдельный тип, которому присваивается имя, что позволяет впоследствии объявлять различные переменные с помощью этого имени, как принадлежащие этому типу. В языке Pascal объявление, по смыслу аналогичное приведенному выше объявлению C, выглядит следующим образом:

```
type Class = (Fresh, Soph, Junior, Senior);
```

Затем можно объявлять переменные этого типа:

```
StudentClass: Class;
TransferStudentClass: Class;
```

Обратите внимание на то, что в определении типа вводится его название, Class, которое затем может быть использовано так же, как название элементарного типа данных, например integer для целых чисел. Определение типа также определяет литералы Fresh, Soph, Junior и Senior, которые могут использоваться в программе таким же образом, как и определенные непосредственно в языке литералы, скажем, 27 или 1. Следовательно, мы можем записать

```
if StudentClass = Junior then ...
```

вместо менее понятной записи

```
if StudentClass = 3 then ...
```

которая потребовалась бы, если бы мы использовали целые числа. Помимо того, при статической проверке типов компилятором могут быть обнаружены такие ошибки, как

```
if StudentClass = Male then ...
```

Основные операции над объектами перечисляемого типа — это операции отношения (равно, больше чем, меньше чем и т. д.), операция присваивания и операции successor и predecessor, которые для данного элемента перечисления определяют соответственно последующий и предыдущий элементы. Операция successor неприменима к последнему элементу перечисления, а операция predecessor соответственно — к первому. Обратите внимание на то, что для перечислений определен полный набор операций отношения благодаря тому, что набор элементов упорядочен при определении типа.

Реализация. Представление перечисления как объекта данных в памяти компьютера достаточно просто: каждому элементу перечисления при выполнении программы сопоставляется неотрицательное целое число (его порядковый номер в последовательности). Поскольку задействовано небольшое количество чисел, причем заведомо неотрицательных, то обычно используются короткие целые числа. В них отсутствует бит, отвечающий знаку, а остальных битов ровно столько, сколько нужно для двоичного представления чисел из требуемого диапазона, как в случае целочисленного поддиапазона. Например, в определенном ранее перечислении Class имеется всего четыре различных элемента, которые во время выполнения программы представляются как 0 = Fresh, 1 = Soph, 2 = Junior и 3 = Senior, поэтому для представления этих элементов в памяти компьютера достаточно двух битов. Следовательно, для переменной типа Class в памяти отводится область длиной два бита. Операции successor и predecessor сводятся просто к добавлению и вычитанию единицы из целочисленного представления элементов перечисления и проверке того, что полученный результат не выходит за допустимые границы.

В С программист может отменить сопоставляемые по умолчанию целочисленные значения и назначить другие произвольным образом, например

```
enum class {Fresh=14, Soph=36, Junior=4, Senior=42}
```

Для определяемого указанным способом представления перечислений реализация элементарных операций не составляет труда, так как можно использовать встроенные в аппаратуру операции над целыми числами. Например, операции отношения между элементами перечисления =, > и < можно реализовать при помощи аппаратных команд сравнения целых чисел.

5.2.3. Логические (булевы) значения

В большинстве языков предусмотрен *логический* (булев) тип данных для представления значений истина и ложь.

Спецификация. Логический тип данных состоит из объектов, которые могут принимать два значения: истина (true) и ложь (false). В Pascal и Ada логический тип данных рассматривается как встроенное в язык перечисление, а именно:

```
type Boolean = (false, true);
```

В этом объявлении определены имена false и true, а также их порядок false < true.

Наиболее распространенными операциями с этим типом данных являются операции присваивания и следующие логические операции:

```
and: Boolean × Boolean → Boolean (И, логическое умножение)
or: Boolean × Boolean → Boolean (включающее ИЛИ)
not: Boolean → Boolean (логическое отрицание или булево дополнение)
```

Иногда включаются и другие логические операции: эквивалентность, исключающее ИЛИ, импликация, nand (not-and), nor (not-or). Сокращенная схема вычисления логических И и ИЛИ обсуждается в разделе 8.2.

Реализация. Объекты логического типа представляются в памяти всего одним битом (если не требуется наличие дескриптора, описывающего тип). Поскольку часто получается так, что один бит в памяти не имеет своего отдельного адреса, то для представления логического объекта используется слово или байт, то есть минимальная адресуемая область памяти. Тогда внутри такой области значения истина и ложь могут быть представлены двумя способами:

- ✦ какой-то определенный бит внутри области памяти (обычно тот же, который отвечает за знак в представлении числа) используется для представления булева значения (ложь = 0, истина = 1), а остальные биты игнорируются;
- ✦ значению ложь отвечает заполнение всей области памяти нулями, а любые другие комбинации битов в этой области соответствуют значению истина.

Поскольку оба эти способа представления могут потребовать большого объема памяти, в языке часто предусматривается специальный способ хранения последовательности битов. Примерами подобного подхода могут служить следующие типы данных: *упакованный массив логических значений* (packed array of Boolean) и *множество* (set) в языке Pascal и *строка битов* в PL/I.

В языке Java логический тип определен явным образом в отличие от C, в котором для этого используются целые числа. Истина выражается любым ненулевым значением, а ложь соответствует нулевому значению. В принципе, такое определение порождает некоторые проблемы. Например, в следующем фрагменте программы на C:

```
int flag;
flag = 7;
```

значение переменной flag устанавливается равным 7, или 111 в двоичном представлении. Однако если вы захотите инвертировать значение переменной flag посредством установки всех битов в его двоичном представлении на противоположные значения, то получится... 1111000, то есть по-прежнему значение будет истина. Аналогично комбинирование значений таких переменных в одно с использованием операции поразрядного логического ИЛИ (|) вместо операции логического ИЛИ (||) или операции поразрядного логического И (&) вместо операции логического И (&&) приводит к таким же проблемам. В языке C всегда более безопасно и надежно использовать для представления логического значения истина целочисленную единицу и не пытаться упаковать несколько булевых значений в одно многобитовое целое число.

5.2.4. Символы

Большинство данных вводится и выводится в символьной форме. Обычно эти данные затем преобразуются к другим типам, но возможность обработки некоторых

символьных данных в их исходной форме также весьма существенна. Последовательности символов (строки символов) обычно обрабатываются как единое целое. Такой тип данных, как строки символов, может быть обеспечен или непосредственно через тип данных *строка символов*, как это сделано в языках ML и Prolog, или при помощи *символьного* типа данных в виде одномерного массива символов, как это сделано в языках Pascal, C и Ada. Строки символов рассматриваются в разделе 5.3.1.

Спецификация. Символьный тип данных предоставляет возможность создания объектов данных, значениями которых может быть какой-либо единственный символ. Множество возможных значений (то есть символов) обычно представляет собой встроенное в язык перечисление, соответствующее стандартным наборам символов, которые поддерживаются аппаратной частью компьютера (например, набор символов ASCII). Порядок символов в этом наборе называется его *схемой сортировки*. Схема сортировки играет существенную роль в операциях отношения, так как определяет алфавитный порядок для строк символов. Поскольку в эту схему включены все символы, то можно упорядочивать строки, содержащие пробелы, точки и другие специальные символы. Кроме операций отношения и присваивания для символьных данных иногда используются операции, которые позволяют определить принадлежность данного символа к какому-либо специальному классу символов: буква, цифра или специальный символ.

Реализация. Символьный тип данных почти всегда поддерживается непосредственно аппаратной частью компьютера, так как объекты этого типа используются для ввода-вывода. Иногда, однако, случается, что какой-либо конкретный набор символов (например, ASCII), включенный в определение языка, не поддерживается аппаратной частью конкретного компьютера. Хотя в обоих наборах могут содержаться одни и те же символы, способы их представления и, следовательно, схемы сортировки могут различаться; в других случаях некоторые специальные символы могут присутствовать в одном наборе символов и отсутствовать в другом. Поскольку символы, введенные в компьютер через устройство ввода-вывода, представлены в том виде, который поддерживается аппаратурой данного компьютера, то в языке должны быть предусмотрены способы преобразования в альтернативные представления либо обеспечена такая реализация операций отношения, в которой учитывается различие схем сортировки. Если же в языке используется тот же способ представления символов, что и в аппаратуре компьютера, то операции отношения обычно реализуются непосредственно через аппаратные или могут быть смоделированы простыми короткими встроенными последовательностями кода.

5.3. Составные типы данных

Типы данных, которые мы рассматриваем в этом разделе, обычно относят к элементарным типам данных. Тем не менее их реализация обычно подразумевает организацию компилятором достаточно сложных структур. Такие типы данных характеризуются наличием нескольких атрибутов.

5.3.1. Строки символов

Строка символов — это объект данных, составленный из последовательности символов. В большинстве языков этот тип играет существенную роль, в первую очередь при вводе и выводе информации.

Спецификация и синтаксис. Можно указать по меньшей мере три различные интерпретации этого типа данных.

1. *Строки фиксированной длины.* Объект данных строка символов может быть объявлен как имеющий определенную фиксированную длину. Значения, присваиваемые этому объекту, могут быть строками символов только этой длины. Присваивание этому объекту строки символов длины, отличной от определенной при его объявлении, приводит к ее усечению в случае, если ее длина больше определенной в объявлении, или к добавлению пробелов в ее конец, если ее длина меньше определенной в объявлении объекта.
2. *Строки переменной длины, не превосходящей заданного максимума.* В программе можно определить некоторую *максимальную* длину строки, а фактические строки могут содержать меньшее количество символов или даже не содержать их совсем (пустые строки). При выполнении программы длина строки может меняться, но если она увеличивается настолько, что становится больше максимальной, то лишние символы отсекаются.
3. *Неограниченные по длине строки.* Объект данных такого типа может содержать строку любой длины, и эта длина может динамически меняться в процессе выполнения программы без каких-либо ограничений.

В языке C ситуация несколько более сложная. В этом языке строки реализуются как массивы символов, но для самих строк символов не предусмотрено специального типа. Однако имеется соглашение, по которому за последним символом строки помещается нуль-символ (`\0`). Транслятор C добавляет нуль-символ в конец каждой строковой константы. Например, в конец строки «В конце этой строки стоит нуль-символ» при компиляции будет добавлен нуль-символ, то есть при размещении символов строки в массиве в него будет добавлен еще один элемент, содержащий нуль-символ. Однако если строка составляется из элементов массива, содержащих символы, явным образом, то программист должен сам добавлять нуль-символ в качестве последнего символа строки.

Первых два способа представления символьных строк позволяют отвести под каждый объект этого типа определенный участок памяти еще при компиляции. Если же длина строк не ограничена, требуется динамически распределять память для таких объектов уже в процессе выполнения программы. Разные способы представления строк предполагают и разные допустимые операции работы с этими объектами.

Существует множество разнообразных операций, применяемых к строкам. Некоторые наиболее важные операции перечислены ниже.

1. *Конкатенация (объединение).* Конкатенация — это операция, объединяющая две строки в одну длинную. Например, если знак `||` обозначает конкатенацию, то результатом выполнения операции `"BLOCK" || "HEAD"` будет строка `"BLOCKHEAD"`[1].

---

[1] Болван. — *Примеч. лит. ред.*

2. *Операции отношений со строками.* К строкам можно применять обычные операции отношения (равно, больше-чем, меньше-чем). Как сказано в разделе 5.2.4, базовое множество символов упорядочено при помощи схемы сортировки. Эту схему можно применить и к строкам, в результате чего получится правило лексикографического (алфавитного) упорядочения: строка A меньше строки B (то есть предшествует строке B), если первый символ в A меньше первого символа в B; если эти символы совпадают, то второй символ A меньше второго символа B и т. д. При этом иногда, если одна из сравниваемых строк длиннее, вторая удлиняется при помощи пробелов, чтобы их длины были одинаковыми.
3. *Выбор подстроки при помощи позиционного индекса.* Обработка строковых данных часто включает в себя выделение из строки непрерывной подстроки. Например, строка может начинаться с некоторого количества пробелов или содержать слова, разделенные пробелами и знаками препинания. Для облегчения работы с подстроками во многих языках предусмотрена операция, позволяющая выделить подстроку по указанным позициям ее первого и последнего символа в исходной строке или по позиции первого символа и длине выделяемой подстроки. Например, оператор языка FORTRAN NEXT = STR(6 : 10) присваивает переменной NEXT подстроку, состоящую из пяти символов строки STR с шестого по десятый включительно. Смысл операции выбора подстроки иногда неочевиден, особенно если разрешено ее использовать с обеих сторон от символа операции присваивания (то есть соответствующая ей функция может возвращать как *r*-значение, так и *l*-значение выделенной подстроки, что позволяет присвоить выбранной подстроке новое значение). Рассмотрим следующее выражение языка FORTRAN:

   ```
   STR(1:5) = STR(I:I+4)
   ```

   которое может быть использовано для того, чтобы передвинуть подстроку, которая начинается с позиции I и имеет длину 5, на место первых пяти позиций этой же строки. Если подстроки, которые задействованы в этом выражении, перекрываются, то требуется большая аккуратность в определении смысла этой операции.
4. *Форматирование ввода-вывода.* Исходное назначение операций над строками символов — помощь в форматировании данных для вывода или в разбиении форматированных данных ввода на более мелкие компоненты. В языках FORTRAN и C, например, предусмотрено большое количество операций, преследующих эти цели.
5. *Выбор подстроки на основе сопоставления с образцом.* Часто бывает так, что точная позиция требуемой подстроки неизвестна, но известно ее положение по отношению к другим подстрокам. Например, возможно, мы захотим получить первый отличный от пробела символ строки, или последовательность символов после десятичной точки, или слово, следующее за словом THE. Операция сопоставления с образцом использует в качестве аргумента образец — специальную структуру данных, определяющую вид искомой подстроки (например, ее длину или то, что она состоит из последовательности десятичных цифр), и, возможно, другие подстроки, примыкающие к искомой (например, десятичная точка после искомой подстроки или последовательность

предшествующих ей пробелов). Вторым аргументом операции сопоставления с образцом является сама строка символов, в которой будет производиться поиск требуемой подстроки, соответствующей заданному образцу. Мы уже встречались с операцией сопоставления с образцом в нашем обзоре языка Perl, когда рассматривали регулярные выражения (см. раздел 3.3.3).

6. *Динамические строки.* Значения строк могут быть как статическими, так и динамическими. Perl является примером языка, в котором допускаются оба варианта. Строка `'$ABC'` статическая, и оператор

   ```
   print '$ABC';
   ```

   напечатает значение этой строки, то есть `$ABC`. Но строка `"$ABC"` динамическая, и в результате выполнения оператора

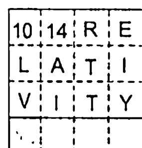
   ```
   print "$ABC";
   ```

   сначала будет вычислено значение строки, которое равно значению строковой переменной `$ABC`, а потом оно уже будет напечатано[1].

Фиксированная длина

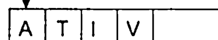
Строки хранятся в блоках фиксированной длины — в одном слове содержится 4 символа, оставшиеся свободными позиции заполняются пробелами

Переменная длина, ограниченная сверху



Текущая и максимальная длины строки хранятся в начале строки

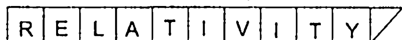
Неограниченная длина, но содержимое размещается в блоках фиксированной длины



Строки хранятся по 4 символа в слове с заполнением пробелами оставшихся свободными байтов

Длина строки хранится в ее начале

Неограниченная длина, но содержимое размещается в блоках переменной длины



Строка хранится как непрерывный массив символов. Заканчивается нуль-символом

Рис. 5.5. Представление строк в памяти

Реализация. Каждый из трех описанных выше способов представления символьных строк по-своему реализуется в памяти компьютера, как показано на рис. 5.5. Представление в памяти отдельных символов обсуждается в разделе 5.2.4. Представление для строк фиксированной длины имеет существенное значение, так как служит основой для хранения упакованного вектора символов (подробнее см. раздел 6.1.5). Для строк переменной длины с заданной максимальной длиной пред-

[1] В Perl символы строк в одинарных кавычках трактуются так, как они в ней заданы, тогда как строки в двойных кавычках — всего лишь более удобная запись операции подстановки в строку символов значений скалярных переменных и переменных массивов. — *Примеч. науч. ред.*

ставление использует дескриптор, указывающий максимальную и фактическую длину строки, содержащейся в данном объекте. Для представления неограниченных по длине строк может использоваться либо связанная цепочка объектов данных фиксированной длины, либо непрерывный массив символов. Второй из упомянутых способов используется в языке С и часто требует динамического управления ресурсами памяти во время выполнения программы.

Обычно аппаратная часть поддерживает только самое простое представление — строки фиксированной длины, а для реализации остальных способов требуется программное моделирование. Такие операции над строками, как конкатенация, выбор подстроки и сопоставление с образцом обычно требуют полного программного моделирования.

5.3.2. Указатели и объекты данных, конструируемые программистом

Обычно вместо того, чтобы включать в определение языка множество различных связанных объектов данных переменного размера, разработчики языка предусматривают возможность конструирования произвольных структур при помощи указателей, связывающих вместе компоненты структуры, представленные разнообразными объектами данных. Для этого язык должен предоставлять некоторые специальные возможности.

1. Элементарный тип данных *указатель* (этот тип также называется *ссылочным*). Объект данных *указатель* содержит ссылку на местоположение другого объекта данных (то есть его *l*-значение) или может содержать также пустой указатель (обычно обозначаемый `nil` или `null`). Указатели — это обычные объекты данных, которые могут быть как простыми переменными, так и компонентами массивов и записей.
2. *Операция создания* объектов данных фиксированного размера, таких как массивы, записи и элементарные объекты. Операция создания выполняет две функции: отводит под новый объект данных блок памяти и возвращает *l*-значение этого объекта, которое затем может быть сохранено как *r*-значение указателя. Операция создания объекта данных отличается от его обычного создания посредством объявления в двух отношениях:
   а) создаваемые при помощи этой операции объекты данных не обязаны иметь имена, так как доступ к ним осуществляется через указатели;
   б) таким способом можно создавать объекты в любом месте программы и в любой момент ее выполнения, а не только при входе в подпрограмму.
3. *Операция разыменования.* Эта операция определена для значений указателей и позволяет получить значение объекта, на который ссылается данный указатель.

Спецификация. Тип данных *указатель* определяет класс объектов данных, значением которых является ссылка на расположения в области памяти некоторых других объектов данных. Объект данных *указатель* можно трактовать одним из двух описанных далее способов.

1. *Указатели могут ссылаться только на объекты одного типа.* Такой подход применяется в C, Pascal и Ada, где используется статический контроль типов и объявления типов. В языке C указатель на любой объект типа List объявляется следующим образом:

   ```
   List *P;
   ```

   Символ * указывает, что типом P является указатель. Тип List в данном случае означает, что значением P может быть *l*-значение любого объекта типа List. Для определения структуры объектов типа List требуется отдельное объявление:

   ```
   struct List {int ListValue; List *NextItem; };
   ```

2. *Указатели могут ссылаться на объекты данных любого типа.* Альтернативой первому способу является разрешение указателю содержать ссылки на различные объекты в различные моменты выполнения программы. Такой подход применяется в языках типа Smalltalk, в которых объекты данных в процессе выполнения программы снабжены дескрипторами типа и в которых реализован динамический контроль типов.

В некоторых языках (например, C и C++) указатели являются объектами данных, которыми можно манипулировать явным образом в программе. В других языках (например, в Java) указатели являются частью скрытых структур данных, управляемых реализацией языка.

Операция создания отводит место в памяти для нового объекта фиксированной длины (и таким образом создает его), а также создает указатель на этот новый объект данных, который можно хранить в объекте данных типа указатель. В языке Ada такая операция называется new. В C эту операцию выполняет системная функция malloc (memory-allocator — распределитель памяти). (Языки C++ и Java упростили C, вернув операцию new как средство распределения памяти.) Рассмотрим подпрограмму, содержащую объявление переменной-указателя P (подобную приведенной выше). При входе в подпрограмму память распределяется только для объекта данных P (память для хранения одного значения указателя). Далее в процессе выполнения подпрограммы может быть создан новый объект данных типа List при помощи следующего оператора:

```
P = malloc(sizeof(List))
```

Поскольку объект данных P был объявлен как указатель только на объекты типа List, смысл этого оператора следующий: создать в памяти блок из двух слов для хранения объекта типа List и *l*-значение этого объекта сохранить в переменной P.

Операция выбора позволяет использовать значение-указатель для доступа к указываемому им объекту данных. Поскольку указатели являются обычными объектами данных, объект данных типа указатель также может быть выбран с помощью обычных механизмов выбора. Например, в C операция выбора объекта, на который ссылается некоторый указатель, обозначается как *. Чтобы выбрать компонент вектора, на который ссылается указатель P, следует написать *P.first. Операция * просто преобразует *r*-значение указателя в *l*-значение. Таким образом, конструкция *P.first обеспечивает доступ к значению, хранящемуся в P, считая, что оно теперь — *l*-значение, и использует его для доступа к компоненту first записи, на которую указывает P.

Реализация. Объект данных типа указатель представляется в виде области памяти, в которой содержится адрес другой области памяти. Это базовый (начальный) адрес блока памяти, представляющего объект данных, на который ссылается указатель. Используются два основных представления значений указателей в памяти.

1. *Абсолютный адрес.* Значение указателя может представлять собой истинный адрес блока памяти, отведенного для объекта данных.
2. *Относительный адрес.* Значение указателя может быть представлено как *смещение* от *базового адреса* некоторого более крупного блока памяти, называемого *кучей*, внутри которого размещается объект данных.

Если значением указателя является абсолютный адрес, то объект данных, созданный операцией new, может быть размещен в любой области памяти компьютера. Обычно эта область выделяется внутри общей кучи. Выбор нужного объекта при помощи абсолютного адреса эффективен, так как значение указателя в данном случае обеспечивает непосредственный доступ к объекту данных, используя встроенные в аппаратуру операции доступа к памяти. Недостатком абсолютной адресации является то, что управление памятью становится более сложным, так как ни один объект данных не может быть перемещен в памяти, пока где-то существует на него указатель, если только значение последнего не изменяется, чтобы отразить новое расположение объекта данных. Восстановление памяти, отведенной под объект данных, после того как он был удален в процессе сбора мусора, также достаточно трудный процесс, так как для каждого объекта данных освобождаемая память восстанавливается в индивидуальном порядке и блоки памяти должны быть возвращены обратно в общий пул доступного свободного пространства памяти. Эти вопросы подробно обсуждаются в главе 10.

Использование относительных адресов в качестве значений указателей требует первоначального выделения блоков памяти, в пространстве которых и выделяется память под объекты данных, последовательно создаваемые операцией new. Для каждого типа данных может выделяться своя отдельная область памяти, а может быть создана одна область, в которой размещаются все объекты всех типов. В любом случае к каждой из таких областей применяются методы *управления кучей*. Если под объекты каждого конкретного типа отводится своя область памяти, то операция new располагает вновь созданные объекты данных в блоках памяти фиксированного размера, что частично упрощает процесс управления памятью. Выбор объектов данных в случае использования относительной адресации обходится дороже, чем при абсолютной адресации, так как для получения фактического (абсолютного) адреса к каждому относительному адресу нужно добавлять базовый адрес области памяти, в которой располагается искомый объект. К преимуществам относительной адресации относится возможность перемещения всей области в целом в любой момент выполнения программы без изменения значений указателя. Например, эта область памяти может быть записана во внешний файл, а затем обратно в какое-нибудь другое место в памяти компьютера. Поскольку сдвиг (относительный адрес — значение указателя) не изменился, то для доступа к какому-либо объекту данных можно использовать прежнее значение указателя, к которому добавляется новый базовый адрес. К дополнительным удобствам использования

относительных адресов можно отнести то, что вся выделяемая под объекты область памяти может рассматриваться как некоторый объект данных, который создается при входе в подпрограмму и используется для размещения объектов, созданных в процессе выполнения этой подпрограммы (а также тех подпрограмм, к которым происходят обращения) при помощи операции new, а затем удаляется при выходе из подпрограммы. В этом случае не требуется восстанавливать в выделенной области память после удаления индивидуального объекта данных, так как вся выделенная под такую область память восстанавливается как единое целое при завершении работы подпрограммы.

Если каждый указатель может содержать ссылку только на объекты данных какого-то одного определенного типа, то, как уже говорилось ранее, становится возможным статический контроль типов. Если же убрать это ограничение, то во время трансляции невозможно будет определить, на объект данных какого типа будет ссылаться указатель при выполнении программы, поэтому нужно будет осуществлять динамический контроль типов. В некоторых языках типы выбранных при помощи указателей объектов данных вообще не проверяются. Во время выполнения программы перед началом выбора объекта также требуется проверять, что значение указателя не равно nil.

Главная проблема при реализации указателей и сконструированных с их помощью объектов данных связана с выделением памяти при выполнении операции создания новых объектов. Поскольку эту операцию можно использовать для создания объектов различного размера в произвольные моменты времени в процессе выполнения подпрограммы, то необходима базовая система управления памятью, способная управлять ее областью, называемой кучей. Для других языковых структур С требуется только основанная на стеке система управления памятью, поэтому добавление указателей и функции динамического выделения памяти malloc к языку требует существенного расширения всей структуры динамического управления памятью во время выполнения программы. Указатели привносят потенциальную опасность накопления в памяти мусора, что может произойти в случае, если все указатели на какой-нибудь созданный объект данных потеряны. Возможно также появление повисших ссылок в том случае, если объект данных, на который ссылался указатель, был уничтожен и освободившееся место в памяти использовано под новый объект данных. Эти вопросы снова будут рассмотрены в главе 6.

5.3.3. Файлы и ввод-вывод

Файл — это структура данных, обладающая двумя особенными свойствами.

1. Обычно она располагается на каком-либо внешнем устройстве хранения (магнитная лента или диск, например), и, следовательно, ее размер может быть значительно больше, чем размеры других типов структур данных.
2. Ее время жизни может значительно превышать время выполнения программы, создавшей ее.

Наиболее распространенным типом файлов являются *последовательные файлы*, но во многих языках также используются *файлы прямого доступа* и *индексно-*

последовательные файлы. Две основные задачи, которые выполняют файлы, достаточно очевидны — это ввод и вывод информации, то есть связь с внешней операционной средой (см. главу 1), и временное хранение данных в том случае, если недостаточно ресурсов быстродействующей памяти. Компоненты файла часто называются *записями*, но в нашей книге мы стараемся не употреблять этот термин в таком смысле, чтобы не вносить путаницу с записями, которые являются специфической структурой данных, обсуждаемой в разделе 6.1.6.

Последовательные файлы

Последовательный файл[1] — это структура данных, состоящая из линейной последовательности компонентов одного типа, переменной длины без ограничения максимального размера (кроме, конечно, естественного ограничения, связанного с размером свободного пространства на имеющемся в наличии внешнем устройстве хранения). В языке Pascal при объявлении файла указываются его имя и тип содержащихся в нем компонентов. Например, следующее объявление:

```
Master: file of EmployeeRec;
```

определяет файл с именем Master, компонентами которого являются объекты данных типа EmployeeRec. Обычно в состав файла не могут входить компоненты переменной длины (то есть, например, недопустимы файлы, состоящие из файлов, или файлы, состоящие из стеков). Кроме того, обычно не допускаются в качестве компонентов файлов структуры, представляющие собой связанные цепочки объектов или включающие значения указателей, потому что по завершении выполнения программы файл остается, а значения указателей лишаются смысла. Когда впоследствии данные считываются из файла, ячейки памяти, на которые ссылаются указатели, вероятно, будут заняты под другие объекты.

Для ввода и вывода данные обычно представляются в символьном виде. Компонентами файла в таком случае становятся отдельные символы, а сам файл называется *текстовым файлом*, в Pascal, например, его обозначают textfile. В большинстве языков для текстовых файлов предусмотрены дополнительные операции ввода-вывода помимо обычных операций над файлами. В текстовом файле, как правило, последовательность символов разбивается на фрагменты, называемые *строками* (lines). Сначала мы рассмотрим обычные последовательные файлы, а затем обсудим несколько специфических особенностей текстовых файлов.

Обычно доступ к файлу осуществляется либо в режиме чтения, либо в режиме записи. В любом случае используется *указатель текущей позиции в файле*, который указывает позицию перед первым компонентом файла, между двумя компонентами или после последнего компонента файла. В режиме записи указатель текущей позиции всегда расположен после последнего компонента, и единственно возможной операцией в таком режиме является присваивание этой позиции нового компонента (иначе говоря, запись очередного компонента), в результате чего файл расширяется на один компонент. В режиме чтения указатель текущей позиции может располагаться в любом месте файла, и единственной возможной опера-

[1] Правильнее было бы сказать файлы с последовательным доступом к их компонентам, так как любой файл представлен в виде линейной последовательности компонентов. — *Примеч. науч. ред.*

цией доступа является доступ к компоненту, следующему непосредственно за указателем, то есть считывание этого компонента. В этом режиме не допускается операций по присваиванию новых компонентов или новых значений существующим компонентам.

Спецификация. Основные операции над последовательными файлами таковы.

1. *Открытие*. Как правило, прежде чем как-либо использовать файл, его нужно открыть. Операция *открытия* получает в качестве параметров имя файла и режим доступа (чтения или записи). Если указан режим чтения, то предполагается, что файл уже существует. Операция *открытия* запрашивает у операционной системы информацию о расположении и свойствах файла, отводит требуемый объем памяти для буферов (см. дальнейшее обсуждение реализации файлов) и устанавливает указатель текущей позиции перед первым компонентом файла. В режиме записи операционная система получает указание создать новый пустой файл или, если файл с таким именем уже существует, уничтожить всю содержащуюся в нем информацию, так чтобы он стал пустым. Указатель текущей позиции устанавливается в начало этого пустого файла.

   Обычно в языке предусмотрен явный оператор *открытия* файла, который следует выполнить перед любыми манипуляциями с содержимым файла. В Pascal, например, имеется процедура `reset`, которая открывает файл в режиме чтения, и процедура `rewrite`, которая открывает файл для записи. Иногда в языке реализована неявная операция *открытия*, которая выполняется при первой попытке прочитать или записать в файл.
2. *Чтение*. Операция *чтения* передает содержимое текущего компонента файла (который обозначен указателем текущей позиции) в определенную переменную программы. Эта передача обычно определена с той же семантикой, что и операция присваивания компонента файла переменной программы.
3. *Запись*. Операция *записи* создает новый компонент в текущей позиции файла (это всегда конец файла) и передает содержимое программной переменной этому новому компоненту. Опять-таки эта передача обычно также определяется как некоторая форма операции присваивания.
4. *Проверка конца файла*. Операция чтения не может быть выполнена, если указатель текущей позиции достиг конца файла. Поскольку длина файла не фиксирована, требуется проводить явную проверку на достижение конца файла, чтобы программа могла предпринять соответствующие действия в этом случае.
5. *Закрытие файла*. Когда обработка файла завершена, его требуется закрыть. Обычно эта операция приводит к передаче сообщения операционной системе о том, что файл следует отсоединить от программы (и сделать потенциально доступным для других программ) и, возможно, освободить область памяти, которая использовалась для обработки файла (буфер и буферные переменные). Часто при завершении работы программы файлы закрываются автоматически, без явного указания со стороны программиста. Тем не менее для явного изменения режима доступа к файлу (от записи к чтению и наоборот) его следует закрыть явным образом и потом открыть в требуемом режиме.

Реализация. В большинстве компьютерных систем используемая операционная система ответственна за реализацию файлов, поскольку они создаются и обрабатываются различными языковыми процессорами и утилитами. Операции над *файлами* в основном реализованы с использованием элементарных команд, предоставляемых операционной системой.

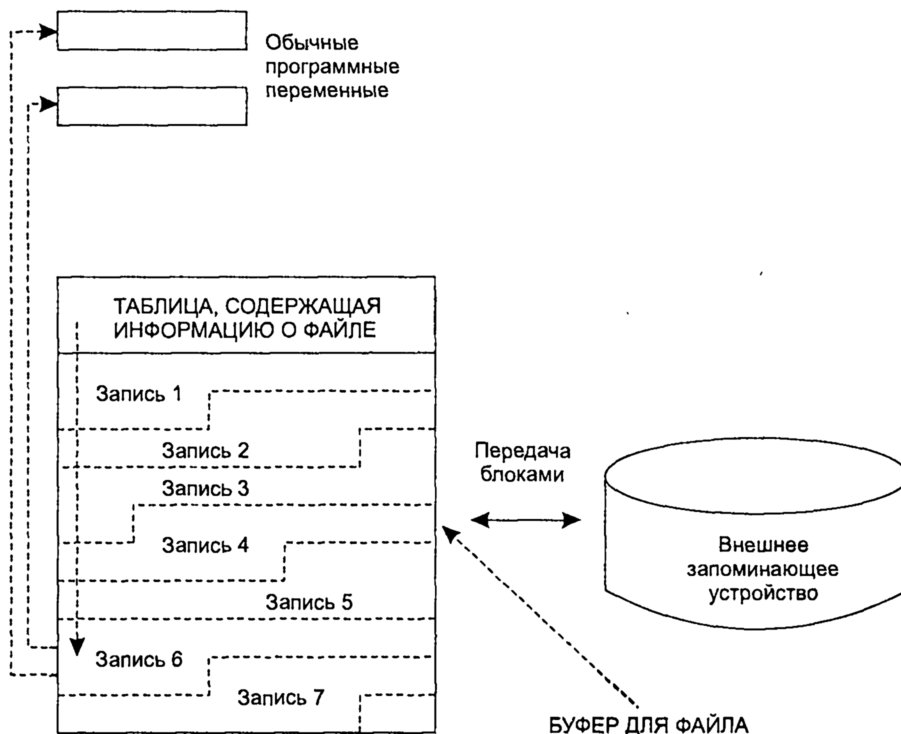


Рис. 5.6. Реализация файла с использованием буфера

С точки зрения языка основная проблема при реализации файлов заключается в выделении памяти для системных данных и буферов, которые используются элементарными системными командами. Обычно, когда в процессе выполнения программы открывается какой-нибудь файл, в памяти отводится место для буферов и таблицы, содержащей информацию о файле. Элементарная команда *открытия* файла операционной системы сохраняет в этой таблице информацию о местоположении и других характеристиках файла. Предположим, файл открыт в режиме *записи*. Когда операция *записи* пересылает компонент, который должен быть добавлен в конец файла, его данные в действительности пересылаются элементарной системной команде *записи*, которая сохраняет их в следующей доступной позиции в буфере, расположенном в памяти компьютера, и изменяет указатели на этот буфер в таблице файловой информации. Никакой фактической передачи данных в файл не происходит до тех пор, пока не будет выполнено достаточное количество операций *записи*, чтобы полностью заполнить блок компонентов в буфере. После этого весь блок компонентов передается из буфера на внешнее запоминаю-

щее устройство (например, диск или магнитную ленту). Следующая последовательность выполняемых в процессе работы программы операций *записи* вновь заполняет буфер до тех пор, пока снова полностью не заполнится его блок компонентов, готовый к передаче на внешнее запоминающее устройство. При считывании файла происходит обратный процесс: данные передаются из файла в буфер в виде больших блоков компонентов, а затем каждая операция *чтения*, выполняемая программой, передает по одному компоненту из буфера в программную переменную. По мере надобности буфер заполняется снова. Эта организация процесса чтения-записи в последовательный файл проиллюстрирована схемой на рис. 5.6.

Текстовые файлы

Текстовый файл (английский термин *textfile* заимствован из языка Pascal) — это файл, состоящий из символов. Текстовые файлы являются основной формой файлов, используемых пользователем для ввода и вывода данных в большинстве языков программирования, так как они могут быть непосредственно распечатаны или введены с клавиатуры. Файлы, содержащие компоненты другого типа, обычно могут быть использованы только для записи и чтения из программы. Текстовые файлы — это всего лишь одна из форм обычных последовательных файлов, и с ними можно производить те же действия, что и с последовательными файлами. Но часто для них определены специальные операции, которые позволяют при вводе автоматически преобразовывать числовые данные (и иногда данные других типов) из символьного вида в их внутреннее представление, используемое при хранении в памяти. Аналогичные операции при выводе позволяют преобразовывать данные из внутреннего представления в символьную форму. Вместе с этими преобразованиями обычно обеспечивается возможность форматирования выводимых данных — разбиение на строки определенной длины с подходящим размещением заголовков, пробелов, интервалов и преобразованием элементов данных к виду, который желателен при печати выводимых данных. Этот *выходной формат* является важной частью реализации операций вывода для текстовых файлов. Аналогичные операции форматирования могут использоваться при вводе либо ввод может осуществляться в *свободном формате* — когда числа, разделенные пробелами, располагаются в любом месте строки ввода. В этом случае операция *чтения* сканирует строку (или строки) ввода и находит требуемые для удовлетворения запроса на чтение числа.

Интерактивный ввод-вывод

Рассмотрим текстовый файл, ассоциированный с интерактивным терминалом, за которым работает программист. При выполнении программы операция *записи* из этого файла интерпретируется как команда отображения символов на экране терминала. Операция *чтения* соответственно представляет команду, которая выполняет запрос на ввод данных с клавиатуры, начало которой обычно характеризуется отображением на экране специального символа-приглашения на ввод данных. При таких установках некоторые аспекты обычной интерпретации последовательных файлов, описанные ранее, несколько изменяются.

1. Файл должен быть открыт *одновременно* в режиме и записи, и чтения, так как обычные операции *чтения* и *записи* при интерактивном вводе-выводе

чередуются. Сначала какие-то данные отображаются на экране, затем какие-то данные ввода запрашиваются и т. д.

2. Буферизация данных операций ввода и вывода ограничена. Во входном буфере редко может накопиться больше одной строки данных, подлежащих обработке. Данные, собранные в выходном буфере, должны быть отображены на экране, прежде чем будет сделан запрос на чтение с терминала.
3. Указатель текущей позиции в файле и проверка конца файла имеют сравнительно небольшое значение. В интерактивном файле отсутствует понятие позиции в том смысле, который обсуждался выше, и у него нет конца, так как программист может продолжать ввод данных неограниченно. Правда, он может использовать специальный управляющий символ, означающий конец ввода с терминала определенной порции входных данных, но обычные представления о проверке конца файла и о завершении процесса обработки файла здесь неприменимы.

Из-за этих существенных различий между интерактивными файлами и обычными последовательными файлами во многих языках возникают проблемы при внедрении интерактивных файлов в структуру ввода-вывода, предназначенную для обычных последовательных файлов[1].

Файлы прямого доступа

В последовательном файле доступ к компонентам должен осуществляться последовательно в порядке их расположения в файле. Хотя обычно имеется некоторый ограниченный набор операций, позволяющий переместить указатель текущей позиции в файле вперед или назад, все же непосредственный доступ к произвольному компоненту, как правило, невозможен[2]. Файл прямого доступа, напротив, организован так, что возможен доступ к произвольному компоненту файла так же, как, например, осуществляется доступ к элементам массивов или записей. Индекс, используемый для доступа к компоненту, обычно называется *ключом* и может быть или целым числом, или каким-нибудь другим идентификатором. Если ключ реализован в виде целого числа, то он очень похож на обычный индекс, используемый для обозначения компонентов массива. Однако в целом реализация файла прямого доступа и операций выбора его компонентов сильно отличается от реализации массива, так как файлы хранятся не в основной памяти, а на вторичных устройствах хранения информации.

Файл прямого доступа реализован как неупорядоченный набор компонентов, с каждым из которых ассоциировано собственное ключевое значение. Исходно файл пуст, и его заполнение осуществляется при помощи операции *записи*, которой в качестве параметров передается компонент, содержимое которого копирует-

---

[1] В современных интерактивных языках программирования файлы ввода с клавиатуры и вывода на экран монитора реализованы как два самостоятельных файла, например, в C стандартные файлы ввода stdin и вывода stdout, в Perl соответственно STDIN и STDOUT и т. д. — *Примеч. науч. ред.*

[2] Здесь важен *непосредственный* доступ, так как в последовательном файле можно получить доступ к любому компоненту, но для этого обычно следует каждый раз устанавливать указатель текущей позиции файла на его начало, а дальше последовательно читать компоненты, пропуская ненужные, пока не будет достигнут требуемый компонент. — *Примеч. науч. ред.*

ся в файл, и ключевое значение, ассоциированное с этим компонентом. Операция *записи* создает новый компонент на внешнем запоминающем устройстве и копирует в него значение переданного ей в качестве параметра компонента. Ключевое значение обычно ассоциируется с местоположением на внешнем запоминающем устройстве созданного компонента путем сохранения пары (ключ, местоположение) в *индексе* — векторе, содержащем подобные пары. Каждая операция *записи*, которая записывает компонент файла с новым ключевым значением, добавляет в индекс новую пару. Но если команда *записи* в качестве аргумента получает значение ключа уже существующего компонента, то содержимое старого компонента заменяется значением компонента, переданного ей в качестве аргумента. Таким образом, запись в файл прямого доступа аналогична присвоению некоторого значения компоненту вектора, где значение ключа является индексом. Операция *чтения* в качестве аргумента получает значение ключа компонента. Затем просматривается индекс и отыскивается пара с таким значением ключа, после чего компонент читается из указанного в паре его местоположения на внешнем запоминающем устройстве.

Индексно-последовательные файлы

Индексно-последовательный файл похож на файл прямого доступа, с тем лишь отличием, что для него предусмотрена дополнительная возможность последовательного доступа к компонентам, начиная с произвольно выбранного. Допустим, например, что для операции *чтения* выбран компонент, значение ключа у которого равно 27. Тогда последующая операция *чтения* может выбрать следующий за ним компонент, причем уже не требуется указывать значение ключа. Такая организация доступа к компонентам файла является промежуточной между файлами прямого доступа и последовательными файлами.

В индексно-последовательном файле, так же как и в файле прямого доступа, используется вектор со значениями ключей (индекс), но расположение ключей в индексе должно быть *упорядочено* в соответствии с их значениями. Когда операция *чтения* или *записи* выбирает некоторый компонент файла с заданным ключом, то соответствующая пара (значение ключа, местоположение компонента) в индексе становится *текущим компонентом* файла, то есть указатель текущей позиции в файле устанавливается на этом компоненте. Чтобы переместиться к следующему компоненту файла, берется следующая пара из индекса, и соответствующий (указанный в этой паре) компонент становится *текущим компонентом*. Таким образом, возможен последовательный доступ к компонентам файла, который не требует больших изменений в организации файлов прямого доступа.

5.4. Обзор языка FORTRAN

FORTRAN — один из первых языков, который широко используется до настоящего времени для инженерных и научных вычислений. За более чем сорокалетнюю историю этого языка он претерпел много изменений, его много раз называли устаревшим и не соответствующим современным задачам, но он по-прежнему существует и развивается.

История. FORTRAN — это первый язык программирования высокого уровня, который получил признание и стал широко применяться. В своем исходном варианте FORTRAN был разработан фирмой IBM в 1957 г. как язык для работы на компьютерах IBM 704. В это время программисты, привыкшие к использованию языка ассемблера, сомневались в возможности использования языков высокого уровня. Наиболее серьезным аргументом была малая эффективность выполнения кода, получающегося в результате трансляции программ, написанных на этих языках. В результате первые версии FORTRAN были ориентированы главным образом на обеспечение эффективности выполнения. Успех этого раннего FORTRAN, связанный главным образом с достижением эффективности выполнения программ на компьютере IBM 704, фактически привел к некоторым затруднениям в дальнейшем развитии языка, о чем мы расскажем далее. Первое стандартное определение языка появилось в 1966 г., а затем в 70-х были внесены существенные изменения, которые привели к появлению FORTRAN 77, и в 90-х — FORTRAN 90.

Обычно при реализации FORTRAN используется стандартная технология создания компиляторов. Для написания программы используется обычный тестовый редактор, а компилятор FORTRAN транслирует программу в исполняемый код. Для объединения подпрограмм, главной программы и набора вспомогательных подпрограмм из стандартных библиотек времени выполнения в единую выполняемую программу используется редактор связей (компоновщик). Завершающим этапом является выполнение программы.

Краткий обзор языка

Как было сказано, разработка языка FORTRAN была подчинена одной главной цели — обеспечению эффективности выполнения программ. Языковые структуры достаточно просты и по большей части не элегантны, но поставленная цель тем не менее достигается. При обсуждении языка FORTRAN можно считать, что FORTRAN 77 и FORTRAN 90 являются совершенно другими языками. В FORTRAN 90 уже добавлены почти все современные возможности управления и представления данных, которые отсутствуют в классическом FORTRAN, поэтому FORTRAN 90 является языком того же уровня, что и языки Pascal и С.

Программа на FORTRAN состоит из главной программы и набора подпрограмм, каждая из которых компилируется отдельно от других. Окончательное объединение оттранслированных программ в выполняемую форму происходит при загрузке. Каждая подпрограмма компилируется в статически размещаемый сегмент кода и запись активации. Во время выполнения программы уже не происходит никаких изменений в распределении памяти, так как распределение памяти осуществляется статически до начала выполнения программы. Некоторые изменения этой модели выполнения сделаны в FORTRAN 90 — в этой версии языка допускается динамическое распределение памяти.

В FORTRAN определено сравнительно мало типов данных: четыре типа числовых данных (целые, вещественные и комплексные числа, а также вещественные с двойной точностью), *булевы* данные (также называемые *логическими*), массивы, строки символов и файлы. Ориентация этого языка на инженерные и научные

вычисления объясняет наличие большого количества встроенных математических функций и арифметических операций. Также предусмотрены операции отношения, булевы операции и простая выборка элементов массивов при помощи индексов. Поддерживаются последовательные файлы и файлы прямого доступа, имеются гибкая система ввода-вывода и большой набор возможностей форматирования.

Структуры управления последовательностью действий включают выражения с обычными инфиксными и префиксными операциями и вызовы функций. Управление последовательностью выполнения операторов в значительной мере опирается на метки и операторы безусловного перехода GOTO, хотя в каждой следующей версии FORTRAN разработчики пытались отойти от этой практики, добавляя вложенные структуры управления. На идеологию языка FORTRAN 66 значительное влияние оказала базовая архитектура аппаратной части компьютера, на котором он был реализован. В FORTRAN 77 были добавлены современные структуры управления (например, оператор условия IF ... THEN ... ELSE), а в FORTRAN 90 эта концепция была разработана до такой степени, что при написании программ на нем теперь можно полностью отказаться от операторов GOTO. При создании FORTRAN 90 появилась концепция *устаревшего свойства*, то есть свойства, которое больше не соответствует современному уровню программирования и от которого следует отказаться в следующей версии этого языка. Примером может служить оператор арифметического IF[1]. Поскольку большая часть свойств FORTRAN 66 на данный момент является устаревшей, ко времени следующего пересмотра стандарта FORTRAN будет вполне современным языком.

В FORTRAN предусмотрены два уровня среды ссылок: глобальный и локальный. Однако в FORTRAN 90 добавлена концепция *вложенных подпрограмм*. Глобальная среда ссылок может быть разбита на несколько общих областей (называемых *COMMON-блоками*; теперь они также находятся в списке устаревших конструкций языка), которые совместно используются несколькими подпрограммами, но только объекты данных допускается использовать таким образом. Параметры в подпрограммы и функции передаются единообразно по ссылке.

5.5. Рекомендуемая литература

Большая часть книг и статей о языках программирования в той или иной мере касается вопросов, связанных с типами и контролем типов (см. ссылки в конце главы 1). Наиболее полно эти вопросы рассмотрены в *Ada Rationale* [57]; там же обсуждаются вопросы представления вещественных чисел с фиксированной и плавающей точками. Трехтомное издание Кнута содержит большое количество материалов по структурам данных и их управлению.

---

[1] Арифметический IF в языке FORTRAN 66 выполнял безусловный переход в зависимости от того, отрицательно, равно нулю или положительно вычисляемое в нем арифметическое выражение: IF(выражение) метка1, метка2, метка3. — *Примеч. науч. ред.*

5.6. Задачи и упражнения

1. Выберите какой-нибудь элементарный тип данных, определенный в хорошо известном вам языке программирования, и проделайте следующее.
   - Опишите множество значений, которые могут содержаться в объектах данных этого типа.
   - Определите, как представлены в памяти значения этого типа данных (в конкретной реализации этого языка, используемой на вашем компьютере).
   - Определите синтаксическое представление для констант этого типа данных.
   - Определите набор операций, определенных для объектов данных выбранного вами типа. Для каждой такой операции приведите ее сигнатуру и синтаксическое представление в языке.
   - Для каждой операции определите, как она реализована: при помощи программного моделирования или непосредственно как одна команда процессора.
   - Опишите какие-нибудь атрибуты объектов данных выбранного типа, отличные от атрибутов этого типа данных.
   - Определите, являются ли какие-нибудь символы или имена операций, определенных для этого типа данных, перегружаемыми. Для каждого перегружаемого имени или символа операции определите, когда (при компиляции или при выполнении программы) определяется специфический смысл каждого использования перегружаемого имени в операторе.
   - Определите, статический или динамический контроль типов применяется для проверки правильности каждого использования каждой операции, определенной для выбранного типа данных.
2. Выберите какой-нибудь хорошо известный вам язык программирования и для какого-либо определенного в нем элементарного типа данных сделайте следующее.
   - Объясните разницу между типом, переменными этого типа и константами этого типа.
   - Приведите пример ситуации, возникающей в процессе выполнения программы, когда объект данных выбранного вами типа существует, но не является ни переменной, ни константой.
   - Объясните разницу между объектами данных этого типа и значениями, которые они могут содержать.
3. Выберите какой-нибудь хорошо известный вам язык программирования и приведите пример элементарной операции, которая:
   - имеет неявный аргумент;
   - приводит к побочным эффектам;
   - не определена для некоторых объектов данных из своей области определения;
   - зависит от предыстории.

4. Приведите формулу для определения максимального количество битов, необходимых для хранения любого целого числа из диапазона $M...N$, где M и N — целые числа, причем $M < N$.
5. Приведите два примера конструкций из какого-либо хорошо знакомого вам языка программирования, в котором контроль типов происходит статически, которые не могут быть статически проверены. Для каждой из них напишите тестовую программу, которая позволит вам выяснить, проверяются ли эти конструкции динамически или так и остаются непроверенными во время выполнения программы.
6. Приведите пример операции из какого-либо языка программирования, которая:
   - реализована непосредственно через аппаратную часть компьютера;
   - реализована как подпрограмма;
   - реализована в виде встраиваемой последовательности кодов.
7. В языках, поддерживающих перечисляемый тип, существует проблема, связанная с перегрузкой буквальных имен в перечислении. При объявлении перечисляемого типа в большой программе вполне вероятна ситуация, когда одно и то же имя будет случайно использовано в различных перечислениях (например, имя `Junior` может быть использовано в определении перечисления `Class` и затем в определении другого перечисления, `Officergrade`). В этом случае ссылка на литерал `Junior` будет неоднозначной. Предложите способ разрешения этой неопределенности, не запрещая перегрузку буквальных имен в перечислении. (Заметим, что в языке Ada такая форма перегрузки разрешена, см. приложение, раздел П.1.)
8. Рис. 5.3 иллюстрирует два способа представления целых чисел с использованием дескриптора типа. В одном случае используется дополнительный объем памяти, но имеется выигрыш в скорости выполнения арифметических операций; в другом, наоборот, за счет увеличения компактности теряется скорость выполнения. Разработайте два аналогичных способа представления целых чисел для вашего компьютера в предположении, что подобный дескриптор занимает по меньшей мере 6 бит. Напишите программы для определения сложения, вычитания, умножения и деления чисел при разработанном вами представлении. Сравните достоинства и недостатки этих двух способов представления.
9. а) Опишите элементарные типы данных, которые встроены в аппаратную часть вашего компьютера. Определите, имеются ли у этих типов данных дескрипторы.

   б) Разработайте полный набор дескрипторов типов для встроенных в аппаратуру типов данных. Каждый дескриптор должен содержать информацию, достаточную для определения местоположения, размера (то есть количество используемых битов) и формата данных, которые он описывает.

   в) Разработайте структуру представления в памяти дескрипторов, логическая организация которых была установлена в предыдущем задании (б).

Чтобы не пришлось решать задачу определения дескрипторов для дескрипторов, сконструируйте эту структуру таким образом, чтобы дескрипторы сами себя описывали (то есть чтобы можно было однозначно определить длину и формат дескриптора только по заданному местоположению первого бита в этом дескрипторе без дополнительной информации).

10. Рассмотрим операцию выбора подстроки `<string variable>(<first char.pos.> : <last char.pos.>)`, как она описана в разделе 5.3.1. Приведите два возможных определения смысла этой операции, когда подстрока используется и как источник, и как объект в операции присваивания:

```
STR(I:J) := STR(K:L)
```

При этом указанные подстроки могут перекрываться.

11. *Конкатенация* — это основная операция над строками символов.

    - Предположим, что строки имеют переменную длину, максимальное значение которой задано в объявлении (см. рис. 5.5). Разработайте для таких строк операцию конкатенации CAT1, которая вызывается с тремя параметрами A, B и C, где A и B — это указатели на блоки памяти, содержащие строки, над которыми надо выполнить эту операцию, а C — результирующий блок, который исходно содержит некоторую другую строку символов. Строка, состоящая из символов строки A и строки B, помещается в блок C (разумеется, с соответствующим дескриптором). Блоки A и B не должны изменяться в результате действия этой операции.
    - Строки неограниченной длины также могут храниться последовательно, с использованием того же способа представления, но максимальная длина должна быть исключена из дескриптора. Разработайте соответствующую структуру представления в памяти таких строк в предположении, что символы могут быть упакованы по четыре в одном слове. Затем разработайте операцию конкатенации CAT2. У нее должны быть два параметра, A и B — объединяемые строки, а результатом ее выполнения является указатель на новый блок памяти, в котором содержится результирующая строка. Предположим, что CAT2 вызывает функцию ALLOCATE(N), которая возвращает указатель на выделенный блок памяти длиной N слов.
    - Разработайте CAT3 — программу, которая объединяет строки, представленные связанными списками, аналогичными изображенным на рис. 5.5.

12. На языке программирования, в котором предусмотрены *указатели* для определяемых программистом объектов данных и операции new и dispose, которые соответственно распределяют память для вновь созданных объектов и освобождают ее при уничтожении ненужных объектов, напишите фрагмент программы, который генерирует *мусор* (с точки зрения управления памятью). Напишите фрагмент программы, который генерирует *повисшие ссылки*. Если какой-либо из этих фрагментов не может быть написан, объясните, почему.

13. Во многих реализациях языка SNOBOL4 множество строк символов, используемых в любой момент выполнения программы, хранится в так называе-

мой *центральной таблице строк*. Она организована как хэш-таблица, в которой каждый элемент является указателем на связанный список блоков памяти. Чтобы проверить, принадлежит ли данная строка X заданному множеству блоков, используется схема двойного хэширования. X хэшируется дважды – как для получения хэш-адреса, который используется в качестве индекса в центральной таблице для получения указателя на соответствующий связанный список блоков памяти, так и для получения *порядкового номера блока*. Каждый элемент в списке блоков состоит из порядкового номера блока и указателя на строку. Элементы заданного списка блоков упорядочены в соответствии с номерами блоков. Для того чтобы определить, хранится ли строка X в блоке, определяемом ее хэш-адресом, в связанном списке блоков ищется блок, порядковый номер которого совпадает или больше порядкового номера блока, содержащего X. В последнем случае X немедленно вставляется в список, а при совпадении порядковых номеров блоков осуществляется посимвольное сравнение X со всеми другими строками в списке, имеющими тот же порядковый номер блока. Запрограммируйте эту схему двойного хэширования, предполагая, что строки хранятся в последовательных блоках с дескриптором длины. Запрограммированная функция должна в качестве входного параметра получать указатель строки; отыскивать эту строку в таблице; вносить ее, если строка не обнаружена, и возвращать адрес найденного вхождения или адрес сформированного вхождения в таблицу.

14. Ввод файлов и проверка конца файла обычно осуществляются до того, как в программе потребуются данные из этого файла или результат проверки, поскольку ввод осуществляется через буфер в блоках, как показано на рис. 5.6. Для интерактивного ввода-вывода более подходящим является альтернативный способ ввода, так называемый *отложенный ввод*. Ввод данных и проверка конца файла происходят, только когда поступает соответствующее требование из выполняемой программы. Разработайте операцию чтения и предложите способ проверки достижения конца файла в случае отложенного ввода из интерактивного файла. Пользователь должен иметь возможность вводить несколько значений за один раз, поэтому при реализации может потребоваться буфер.

Глава 6. Инкапсуляция

При написании больших программ программист почти неизбежно сталкивается с необходимостью разработки и реализации новых типов данных. Например, при написании программы, которая регистрирует зачисленных в университет студентов и создает списки групп, один из первых этапов может заключаться в том, чтобы определить тип объекта данных, который соответствовал бы одной *группе* какого-либо *курса*. В таком объекте данных должна содержаться следующая информация: имя преподавателя, номер аудитории, максимальное количество слушателей и т. д. Эти сведения могут рассматриваться как атрибуты типа данных *группа курса*, так как они предполагаются постоянными в течение всего времени жизни этого объекта. Этот объект должен содержать также список записавшихся студентов[1], которых можно рассматривать как компоненты этого объекта данных. Конкретный объект данных этого типа может представлять конкретную группу студентов, записавшихся на конкретный курс, с указанием значений всех атрибутов и заполненным списком студентов. Далее должен быть определен некоторый набор операций, который обеспечил бы возможность осуществлять базовые манипуляции с объектами данных *группа*: создание нового объекта, включение студента в существующий объект, уничтожение объекта и т. д. Вся эта деятельность по определению объекта *группа* представляет собой разработку спецификации для абстрактного типа данных *группа* (то есть разработку необходимых атрибутов и операций).

Реализация объектов типа *группа* происходит на следующем этапе конструирования программы. Выбор конкретного вида реализации для объектов этого типа должен быть сделан на основе типов данных, определенных в языке; также могут быть использованы другие абстрактные типы данных, например *студент*, *преподаватель* и т. д. Так, имя преподавателя может быть реализовано как строка длиной не более 10 символов или как целое число (идентификационный номер, присвоенный преподавателю), в то время как список зачисленных на курс студентов может быть представлен одномерным массивом целых чисел (идентификационные номера студентов). Когда реализация типа данных *группа* определена, можно переходить к реализации операций над объектами этого типа, например, в виде подпрограмм, аргументами которых являются эти объекты.

---

[1] В американских университетах студенты могут сами выбирать курсы, которые они желают прослушать, заранее записавшись на них. — *Примеч. науч. ред.*

Если программа достаточно велика, над другими ее частями могут работать другие программисты; но теперь они также могут использовать тип данных *группа*. В их распоряжении находятся определенные ранее представления этого типа и подпрограммы, позволяющие манипулировать *группами* (объектами данных типа *группа*), используя спецификацию на подпрограммы, совершенно не заботясь о том, как они на самом деле реализованы. Для этих программистов добавление абстрактного типа данных равносильно добавлению нового типа в определение языка. Например, в большинстве языков определены элементарный тип целочисленных объектов и ряд операций над ними, в результате чего программист может пользоваться этими объектами и операциями, не задумываясь о деталях представления целых чисел в виде комбинаций битов. Точно так же теперь (в пределах данной программы) имеется тип данных *группа* более высокого уровня и ряд операций, определенных для объектов этого типа, что позволяет программисту не вникать в детали реализации *группы* в виде массивов, записей, строк символов и тому подобных компонентов.

Одна из задач, стоящих в наше время перед разработчиком языка, — добиться того, чтобы различия между формами представления типов данных не были заметны для программиста, использующего эти типы данных при создании программ. Иначе говоря, синтаксические структуры языка, используемые для работы с целочисленными переменными и с переменными типа *группа*, не должны различаться. Подобным же образом сигнатуры операций целочисленного сложения `сложить : целое × целое → целое` и добавления студента в объект *раздел курса* `добавить_к_курсу : студент × группа → группа` должны обладать похожим синтаксисом и иметь сопоставимую семантику.

Для того чтобы обеспечить программистам возможность создания новых типов данных и определения для них операций, существуют четыре механизма.

1. *Структурированные данные*. Практически во всех языках предусмотрена возможность создания сложных объектов данных из элементарных объектов, входящих в определение языка. Совокупность однородных объектов может быть определена как массив, список или множество. Совокупность неоднородных объектов можно создать при помощи записей.
2. *Подпрограммы*. Программист может создавать подпрограммы, реализующие желаемую функциональность нового типа данных; однако корректность использования этого типа почти целиком зависит от программиста, так как автоматическая поддержка со стороны языка программирования весьма незначительна.
3. *Объявления типов*. В языке предусмотрена возможность определения новых типов и операций над ними. Понятие *абстрактных типов данных* (см. раздел 6.2), позволяющее программисту создавать новые типы данных, реализовано в языках типа Ada и C.
4. *Наследование*. Возможности определения новых типов данных и операций над ними значительно расширились благодаря использованию понятий объектно-ориентированного программирования и наследования. Кроме того, появилась возможность автоматической проверки правильности использования этих типов. Концепция наследования рассматривается отдельно в главе 7.

6.1. Структурированные типы данных

Структура данных — это такой объект данных, который содержит другие объекты данных в качестве своих элементов или компонентов. В предыдущей главе мы рассматривали понятия объектов и типов данных на простых примерах элементарных типов данных. В этом разделе рассматриваются аналогичные вопросы, но уже в более сложном случае структур данных; в частности, обсуждаются такие важные типы данных, как массивы, записи, стеки, списки и множества.

6.1.1. Структурированные объекты данных и типы данных

Объект данных, сконструированный как совокупность других объектов данных (*компонентов*), называется структурированным объектом данных, или *структурой данных*. Компонентом этой структуры может быть элементарный объект данных или структура данных (например, компонентом массива может быть число, а может быть и символьная строка, запись или другой массив).

Многие вопросы и концепции, касающиеся структур данных, мы уже обсуждали при рассмотрении элементарных типов данных; иначе говоря, эти концепции одинаковы для структур и для элементарных типов. Как и в случае элементарных типов данных, некоторые структуры данных определяются программистом, а некоторые определяются системой в процессе выполнения программы. Связывание структуры данных со значениями, именами и областями памяти несколько более сложно, чем для элементарных типов данных.

Центральной задачей спецификации и реализации структурированных типов данных является определение компонентов этой структуры и взаимоотношений между ними таким образом, чтобы процесс выборки конкретного компонента из этой структуры был прост и однозначен. Требуют особого внимания и вопросы распределения памяти при выполнении многих операций над структурами данных, которые не возникают в случае элементарных типов данных.

6.1.2. Спецификация типов структур данных

Основные атрибуты структур данных следующие.

1. *Количество компонентов.* Структура данных может иметь *фиксированный размер*, если количество входящих в нее компонентов неизменно в течение времени жизни этой структуры, либо *переменный размер*, если это количество динамически изменяется. Для типов структур данных переменного размера обычно определены операции вставки и удаления компонентов. *Массивы* и *записи* обычно приводятся в качестве примеров типов структур данных фиксированной длины; *стеки*, *списки*, *множества*, *таблицы* и *файлы* являются примерами типов структур данных переменного размера. Программисты часто используют *указатели*, позволяющие явным образом связывать объекты данных фиксированного размера для получения объектов данных переменной длины.

2. *Тип каждого компонента.* Структура данных является *однородной* (*гомогенной*), если все ее компоненты одного типа. В противном случае она называется *неоднородной* (*гетерогенной*). Массивы, множества и файлы обычно однородны, в то время как записи и списки, как правило, неоднородны.
3. *Имена, используемые для выбираемых компонентов.* Тип, определяющий структуру данных, должен быть снабжен *механизмом выборки*, позволяющим идентифицировать и выбирать отдельные компоненты структуры данных. Для массива именем отдельного компонента может являться целочисленный индекс или последовательность индексов; в случае записи именем обычно служит определенный программистом идентификатор; так же может определяться имя для компонента таблицы. В некоторых структурах данных, например в стеках и файлах, в каждый конкретный момент времени возможен доступ только к определенным компонентам (например, к верхнему или текущему элементу) и определены операции, при помощи которых можно изменять доступный в данный момент компонент.
4. *Максимальное количество компонентов.* Для структуры данных переменной длины, например для стека, может быть указан максимальный размер (максимальное число входящих компонентов).
5. *Организация компонентов.* Наиболее распространенным видом организации компонентов в структуре данных является их простая линейная последовательность. Примерами структур данных с линейной организацией являются векторы (одномерные массивы), записи, стеки, списки и файлы. Но массив, список и запись, однако, обычно расширяются до многомерных форм: многомерные массивы, записи, компонентами которых являются другие записи, и списки, состоящие из списков. Эти расширенные формы могут рассматриваться как отдельные типы или просто как базовые последовательные типы, в которых компонентами являются структуры данных того же типа. Например, двухмерный массив (матрицу) можно рассматривать как отдельный тип (как в языке FORTRAN `A(i.j)`) или как вектор векторов (как в C `A[i][j]`), то есть вектор, компонентами которого (строками или столбцами) являются также векторы. Для записей тоже имеются варианты — альтернативные множества компонентов, из которых только один включен в каждый объект данных этого типа.

Операции над структурами данных

Спецификация области определения и диапазона значения для операций над структурами данных задается почти так же, как и в случае элементарных типов данных. Большое значение имеют следующие новые классы операций, специфические для структур данных.

1. *Операции выборки компонента.* Обработка структур данных часто требует поочередной выборки компонентов этой структуры. Существует два типа операций выборки, посредством которых осуществляется доступ к компонентам структуры данных для их последующей обработки другими операциями: *произвольная выборка*, при которой можно получить доступ к произвольному компоненту структуры, и *последовательная выборка*, при которой

компоненты перебираются в заранее определенной последовательности. Например, при обработке вектора операция индексации позволяет выбирать произвольный компонент вектора (например, V[4]), а ее использование совместно с циклами for и while реализует последовательный выбор компонентов:

```
for I := 1 to 10 do ... V[I] ...;
```

2. *Операции над всей структурой данных.* Такие операции в качестве аргументов используют структуру данных целиком, а их результатом также является некоторая структура данных. В большинстве языков предусмотрено некоторое ограниченное количество таких операций (например, сложение двух массивов, операция присваивания одной записи другой, объединение множеств). В таких языках, как APL и SNOBOL4, представлен широкий спектр операций над структурами данных, так что программисту редко приходится выбирать отдельные компоненты этих структур для обработки.
3. *Вставка/удаление компонентов.* Операции, которые меняют количество компонентов в структуре данных, оказывают большое влияние на способы представления и управления ресурсами памяти для структур данных. Более подробно об этом мы расскажем в следующем разделе.
4. *Создание/уничтожение структур данных.* Операции по созданию и уничтожению структур данных также оказывают большое влияние на способы управления ресурсами памяти для структур данных.

Выборку (или *доступ*) компонента или значения некоторого объекта данных следует отличать от близкой по смыслу операции *ссылки* на нее (см. подробное обсуждение этого вопроса в главе 9). Обычно объект данных имеет некоторое имя (например, вектор с именем V). Когда мы в программе пишем V[4], подразумевая, что нам нужен четвертый компонент вектора V, на самом деле имеет место следующая двухступенчатая последовательность действий: сначала выполняется *операция ссылки*, а затем — *операция выборки*. Операция ссылки определяет текущее местоположение объекта под именем V (то есть *l*-значение этого объекта) и в качестве результата возвращает указатель на местоположение объекта (то есть всего вектора, обозначенного этим именем). Операция выборки по этому указателю и по указанному индексу 4, которым обозначен нужный компонент вектора, находит и возвращает указатель на местоположение в памяти этого конкретного компонента. В этом разделе мы будем заниматься только операцией выборки. Обсуждение операции ссылки (которая может оказаться гораздо более сложной и дорогостоящей, чем операция выборки) мы отложим до главы 9, где будут подробно рассмотрены проблемы определения имен, правила, определяющие области видимости, и среды ссылок.

6.1.3. Реализация типов структур данных

Задачи, которые необходимо решить для реализации структурированных типов данных, аналогичны тем, что возникают при реализации элементарных типов данных. Кроме того, здесь имеются два дополнительных момента, которые оказывают влияние на выбор способа представления структур данных в памяти при реализа-

ции языка программирования: эффективная выборка компонентов структуры данных и эффективное управление структурой как единым целым.

Представление структур данных

Представление структур данных в памяти включает в себя:

1) хранение отдельных компонентов структуры;
2) хранение необязательного *дескриптора*, в котором хранятся некоторые или все атрибуты структуры.

На рис. 6.1 изображены две основные схемы представления.

1. *Последовательное представление*, когда структуры данных хранятся в одном непрерывном блоке памяти, содержащем и дескриптор, и компоненты.
2. *Связанное представление*, когда структуры данных хранятся в нескольких отдельных блоках памяти, связанных между собой при помощи указателей. Указатель, связывающий блок А с блоком В, называется *связью*. Он представлен адресом первой позиции блока В, который хранится в специально отведенном для него месте в блоке памяти А.

Последовательное представление используется для структур фиксированного размера и иногда для однородных структур данных переменного размера, например для символьных строк или стеков. Связанное представление обычно используется для структур переменного размера, например для списков. В следующих разделах рассматриваются некоторые вариации последовательного и связанного представлений.

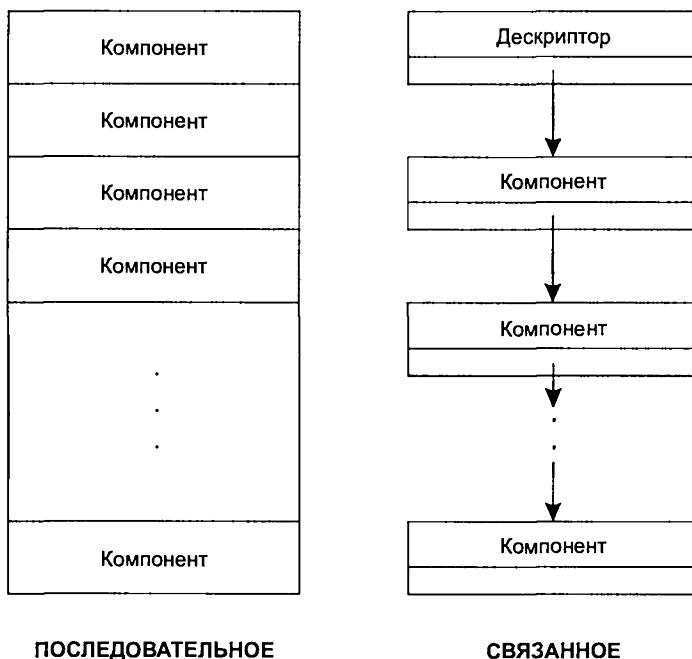


Рис. 6.1. Способы представления линейных структур данных в памяти

Память компьютера, как правило, структурирована в виде простой последовательности байтов. Аппаратные операции, используемые для обеспечения доступа к компонентам структур данных, обычно ограничиваются операциями индексации с использованием индексных регистров, которые позволяют эффективно реализовать доступ к компонентам векторов, представленных в памяти последовательным способом. В большинстве компьютеров аппаратная часть не поддерживает использование дескрипторов для структур данных, манипуляцию структурами при их связанном представлении, распределение памяти для структур данных и управление внешними файлами. Таким образом, структуры данных и определенные для них операции обычно реализуются при помощи программного моделирования в реализации виртуального компьютера языка программирования. В этом заключается существенное отличие структур данных от элементарного типа данных, для которых, как правило, возможно аппаратное представление объектов и операций над этими объектами.

Реализация операций над структурами данных

Выборка компонентов — одна из наиболее важных операций при реализации большинства структур данных. Необходимо обеспечить эффективность выполнения этой операции как для произвольного, так и для последовательного способа. Например, выборка компонента A[I] для произвольного I-го элемента вектора A[I] должна быть достаточно эффективной. Также весьма желательно, чтобы последовательная выборка элементов A[1], A[2],... вектора была более эффективной, чем просто аналогичная последовательность операций произвольного выбора элементов вектора. Эти два основных способа выборки компонентов реализуются по-разному для последовательного и связанного представлений объекта в памяти.

Последовательное представление. Произвольная выборка компонентов часто подразумевает вычисление адреса с использованием *формулы доступа*: базовый адрес + смещение. Относительное местоположение выбранного компонента в последовательном блоке называется *смещением*, а местоположение начала всего этого блока в памяти называется его *базовым адресом*. Формула доступа по имени или индексу требуемого компонента (например, по целочисленному индексу компонента массива) определяет, как вычислять смещение для данного компонента. Затем полученное смещение добавляется к базовому адресу для получения фактического адреса требуемого компонента в памяти компьютера, как показано на рис. 6.2. Например, в языке С массив char A[10] представлен в памяти последовательным способом: A[0], A[1], ..., A[9]. Адрес компонента A[1] будет состоять из базового адреса массива A (в данном случае это *l*-значение элемента A[0]) со смещением на 1. Вообще говоря, в языке С для массивов типа char адрес элемента A[I] массива A будет равен *l-значению* (A)+I. Обратите внимание на то, что такой способ вычисления может быть очень эффективным; если в качестве индексов используются константы, то транслятор может вычислить формулу доступа для компонента еще во время компиляции и сгенерировать код для непосредственного доступа к нему.

Для однородной структуры, например для массива, представленного в памяти последовательным способом, выборка последовательности компонентов может осуществляться следующим образом.

1. Для выбора первого компонента последовательности используется формула доступа, то есть вычисляется сумма базового адреса и смещения компонента.
2. Для перемещения к следующим компонентам последовательности к местоположению текущего компонента добавляется размер этого компонента. В случае однородной структуры размеры всех компонентов одинаковы, поэтому адрес каждого компонента последовательности может быть найден путем добавления некоторой постоянной величины к адресу предыдущего компонента.

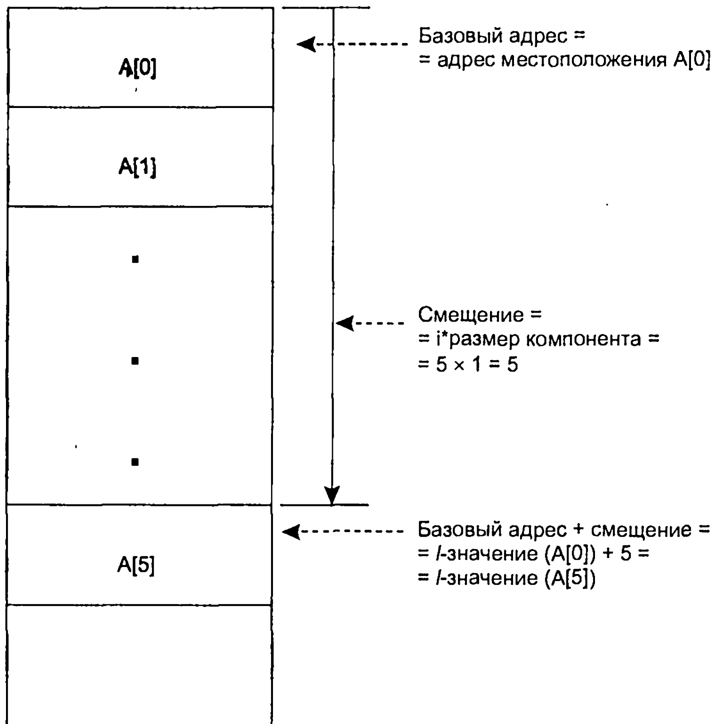


Рис. 6.2. Доступ к компонентам массива типа char в языке С

Связанное представление. Произвольная выборка компонента из связанной структуры подразумевает следование по цепочке указателей от первого блока, представляющего эту структуру в памяти, к искомому компоненту. Для такого алгоритма выборки местоположение в блоках каждого компонента указателя на следующий блок должно быть известно. Выборка последовательности компонентов происходит путем выбора первого компонента и следования по цепочке указателей к последующим компонентам.

Представления структур данных иногда расширяются, чтобы включить в себя определяемые системой структуры, которые позволяют эффективно осуществлять выборку компонентов. Например, произвольная выборка компонентов обыкновенного последовательного файла затруднена, если в представлении этой структуры

отсутствуют индексы (см. раздел 5.3.3). Другим примером являются хэш-таблицы для представления множеств.

Управление ресурсами памяти и структуры данных

Время жизни любого объекта данных отсчитывается от того момента, когда происходит связывание объекта с конкретной областью памяти (то есть когда этому объекту отводится некоторый блок (или блоки) памяти и инициируется его размещение в памяти). Заканчивается время жизни, когда эта связь между объектом и блоком памяти разрушается. В случае объектов данных переменного размера для каждого компонента время жизни определяется индивидуально. Оно отсчитывается от момента создания компонента внутри структуры данных и заканчивается, когда компонент удаляется из этой структуры.

В момент создания объекта данных (то есть в начале времени жизни) создается также *путь доступа* к нему, который необходим для того, чтобы имеющиеся в программе операции могли обращаться к этому объекту данных. Создание пути доступа происходит или через связывание объекта данных с некоторым идентификатором, который становится *именем* этого объекта в некоторой среде ссылок (см. главу 9), или через указатель на структуру в некоторой другой уже существующей и доступной структуре. В этом последнем случае наш объект данных становится компонентом некой старой структуры.

В течение времени жизни объекта могут быть созданы дополнительные пути доступа к нему, например, если этот объект передается в подпрограмму как параметр или если для него создаются новые указатели. Пути доступа могут быть уничтожены различными способами, в частности посредством присвоения указателю нового значения или при выходе из подпрограммы. Вследствие этого будет потеряна связь объекта со средой ссылок. Таким образом, в течение времени жизни объекта может существовать несколько путей доступа к нему.

Несоответствие между временем жизни объекта и временем жизни путей доступа к нему порождает две основные проблемы управления ресурсами памяти.

1. *Мусор*. Когда все пути доступа к объекту данных разрушены, но сам объект продолжает существовать, говорят, что он стал *мусором*. Мало того что к этому объекту данных другие программные конструкции больше не имеют доступа, он просто становится бесполезен. Поскольку связывание объекта с некотором блоком памяти осталось в силе, этот блок остается занят и недоступен для использования.
2. *Повисшие ссылки*. Повисшая ссылка — это путь доступа, который продолжает существовать после истечения времени жизни ассоциированного с ним объекта данных. Путь доступа обычно указывает местоположение объекта данных (то есть начальную позицию того блока памяти, в котором содержится объект). После завершения времени жизни объекта этот блок восстанавливается для дальнейшего использования и впоследствии может быть отведен под какой-либо новый объект данных. Тем не менее восстановление блока памяти для дальнейшего использования не обязательно влечет за собой разрушение всех существующих ссылок (то есть путей доступа), и, следовательно, они могут продолжать существовать как *повисшие ссылки*.

Повисшие ссылки представляют собой особенно серьезную проблему для управления ресурсами памяти, так как они могут нарушить целостность всей структуры распределения памяти при выполнении программы. Например, присваивание несуществующему объекту значения через повисшую ссылку может изменить содержимое памяти, уже распределенной под другой объект данных совершенно другого типа (что нарушит безопасность структуры контроля типов). Или такое присваивание может изменить вспомогательные данные (такие, как ссылку на список свободных блоков памяти), которые временно были помещены системой в тот блок памяти, на который указывала повисшая ссылка. Тем самым нарушится целостность системы управления ресурсами памяти.

Наличие мусора — не такая серьезная проблема, тем не менее она причиняет некоторые неудобства. Объекты данных, которые стали мусором, занимают определенный объем памяти, который мог бы использоваться для других целей. Поскольку в любом компьютере, даже в больших машинах с оперативной памятью, составляющей многие мегабайты, ее объем все же конечен, неограниченное накопление мусора может вызвать преждевременную остановку выполнения программы из-за нехватки памяти для выполнения операций. Структуры управления ресурсами памяти, предназначенные для решения проблем с повисшими ссылками и мусором, обсуждаются в главе 10.

6.1.4. Объявления структур данных и контроль типов

Основные понятия и проблемы, связанные с объявлениями структур данных и контролем их типов, аналогичны тем понятиям и проблемам, которые уже обсуждались в связи с элементарными типами данных. Тем не менее в случае со структурами данных все несколько усложняется, так как они являются более сложными и поэтому требуется определять большее количество атрибутов. Например, объявление языка C

```
float A[20];
```

расположенное в начале подпрограммы P, определяет следующие атрибуты массива A:

1) тип данных — массив;
2) количество размерностей — 1;
3) количество компонентов — 20;
4) индексы представляют собой целые числа от 0 до 19;
5) тип данных каждого компонента — вещественный.

Объявление всех этих атрибутов позволяет выбрать последовательное представление для массива A, а также выбрать подходящую формулу доступа к любому компоненту A[I] массива A, которая может быть вычислена во время компиляции (см. следующий раздел), хотя фактически массив A создается только при входе в подпрограмму P во время выполнения программы. Если бы не было этого объявления, атрибуты массива A пришлось бы определять динамически во время выполнения программы, в результате чего представление массива в памяти и алгоритм доступа к его компонентам были бы гораздо менее эффективными.

Проверка типов для структур данных — несколько более сложная процедура, так как приходится принимать во внимание операции выбора компонентов. Здесь возникают две основные проблемы.

1. *Существование выбранного компонента.* Может возникнуть такая ситуация, что все аргументы для операции выборки имеют правильный тип, но искомый компонент попросту не существует в этой структуре данных. Например, операция индексации, которая используется при выборе компонента массива, может получить значение индекса, выходящее за границы, допустимые для этого массива (то есть будет получено неправильное l-значение для компонента массива). Это не создаст проблемы с проверкой типов при условии, что операция выбора при этом не выполняется, а просто появляется сообщение об ошибке и генерируется исключение (например, ошибка диапазона индексации). Однако если динамический контроль типов отключен в целях повышения эффективности и исключение не генерируется, то результат операции выбора почти всегда немедленно становится аргументом какой-либо другой операции. Если при этом операция выбора выдала неверное l-значение, то результат равносилен ошибке определения типов. Операция, которая использует результат операции выбора, получает в качестве аргумента неверное l-значение (адрес блока памяти, в котором могут содержаться данные другого типа, выполняемый код и т. д.). Операции выбора компонентов, которые используют формулу доступа для вычисления смещения искомого компонента в непрерывной последовательности блоков памяти, особенно подвержены этой ошибке; в этом случае во время выполнения программы часто требуется проверка существования компонента до вычисления формулы доступа, используемой для определения его точного l-значения.
2. *Тип выбранного компонента.* Последовательность выбора может определять сложный путь доступа через структуру данных к искомому компоненту. Например, в языке C оператор

   ```
   A[2][3].link → item
   ```

   выбирает из записи содержимое компонента с именем `item` с помощью указателя, хранящегося в компоненте `link` другой записи, которая, в свою очередь, является элементом второй строки и третьего столбца массива `A`. Для выполнения статического контроля типов должна существовать возможность определения во время компиляции типа компонента, выбираемого любым допустимым селектором подобного составного типа. Как было отмечено ранее, вообще нельзя заведомо предполагать, что выбранный компонент будет существовать в нужный момент при выполнении программы. Статический контроль типов может гарантировать только то, что, если компонент существует, его тип — правильный.

6.1.5. Векторы и массивы

Векторы и массивы — это наиболее распространенные типы структур данных, имеющиеся в языках программирования. *Вектор* — это структура данных, состоящая из фиксированного количества компонентов одного типа, организованных в виде

простой линейной последовательности. Компонент вектора выбирается путем указания *индекса*, который является целочисленным значением (или элементом перечисления), задающим местоположение компонента в этой последовательности. Векторы называются также *одномерными* или *линейными* массивами. В *двухмерном массиве*, или *матрице*, компоненты организованы в форме прямоугольной таблицы, состоящей из строк и столбцов. Для выбора компонента матрицы необходимы два индекса: номер столбца и номер строки, на пересечении которых расположен компонент. Многомерные массивы с размерностью больше двух определяются таким же образом.

Векторы

Атрибуты вектора таковы.

1. *Количество компонентов* обычно указывается неявным образом путем задания последовательности диапазонов изменения индексов, по одному диапазону на каждую размерность.
2. *Тип данных для каждого компонента*, который в данном случае одинаков для всех компонентов.
3. *Список значений индексов, используемых для выбора компонентов*, обычно задается в виде набора целых чисел, первое из которых соответствует первому компоненту, второе — второму компоненту и т. д. Он может задаваться как диапазон значений, например `-5..5`, или определяться только верхней границей диапазона, а нижняя задается по умолчанию, например `A(10)`.

Типичным примером объявления вектора является следующее объявление вектора в языке Pascal:

```
V: array [-5..5] of real;
```

определяющее вектор из одиннадцати компонентов, каждый из которых представлен вещественным числом, причем компоненты вектора выбираются с помощью индексов -5, -4, ..., 5. В языке С объявление выглядит несколько проще:

```
float a[10];
```

Здесь объявляется массив из десяти компонентов с индексами от 0 до 9.

В тех языках, которые позволяют в объявлении массива указывать диапазон значений индексов, этот диапазон не обязательно начинается с 1, как показано в предыдущем примере для массива `V` из языка Pascal. Этот диапазон даже не обязательно должен быть подмножеством целых чисел; он может быть перечислением (или последовательностью перечислений), например:

```
type class = (Fresh,Soph,Junior,Senior);
var ClassAverage: array [class] of real;
```

Операции над векторами. Операция выбора компонента вектора называется *индексацией*; обычно синтаксически она записывается как имя вектора, за которым следует значение индекса искомого компонента (например, `V[2]` или `ClassAverage[Soph]`). Но, в принципе, значение индекса может быть и вычисляемым значением, что приводит к заданию выражения, которое вычисляет индекс компонента (например, `V[I + 2]`). Как уже говорилось, операция индексации возвращает *l*-значение искомого компонента, то есть его местоположение. Если на самом деле требуется

содержимое этого компонента, то есть его *r*-значение, то по найденному адресу (*l*-значению) получить это значение не составляет труда.

Другие операции над векторами включают их создание и уничтожение, присвоение значения компонентам вектора и арифметические операции над парами векторов одинаковой размерности, например их сложение (при этом складываются значения соответствующих компонентов). Поскольку векторы имеют размерность, вставка и удаление компонентов не допускаются; может меняться только значение компонента. В большинстве языков имеется довольно ограниченный набор операций с векторами, но APL содержит большое количество операций, позволяющих, в частности, осуществлять декомпозицию вектора на компоненты и создавать новые векторы различными способами.

Реализация. Однородность компонентов и фиксированная размерность векторов упрощают их хранение и доступ к отдельным компонентам. Однородность означает, что размер и структура каждого компонента одинаковые, а размерность вектора подразумевает инвариантность количества компонентов и их местоположения в течение всего времени его жизни. Подходящим способом реализации такой структуры является последовательное представление компонентов в памяти, как показано на рис. 6.3. В эту структуру также может быть включен дескриптор, описывающий некоторые или все атрибуты вектора, особенно в том случае, если они окончательно определяются только во время выполнения программы. Верхнюю и нижнюю границы диапазона значений индексов (которые не требуются для доступа к компонентам) обычно сохраняют в дескрипторе на случай, если потребуется проверка соответствия индекса объявленному диапазону. Другие атрибуты обычно не хранятся в дескрипторе во время выполнения программы; они нужны только во время компиляции для контроля типов и определения способа представления вектора в памяти.

Если начинать с первого компонента вектора, то для доступа к I-му компоненту следует пропустить I - 1 компонентов. Если размер каждого компонента E, то следует пропустить (I - 1) × E минимально адресуемых областей памяти (ячеек). Если LB — нижняя граница диапазона индексов, то количество пропускаемых компонентов будет I - LB или (I - LB) × E ячеек памяти. Если первый компонент вектора начинается с ячейки с адресом α, то для *l*-значения компонента вектора получается следующая *формула доступа*:

```
l-значение(A[I]) = α + (I - LB) × E
```

Это можно переписать как

```
l-значение(A[I]) = (α - LB × E) + (I × E)
```

Обратите внимание на то, что когда для вектора выделено место в памяти, выражение (α - LB × E) становится постоянной величиной (обозначим ее K) и формула доступа упрощается:

```
l-значение(A[I]) = K + I × E
```

Для языков типа FORTRAN величина K является постоянной и может быть вычислена во время компиляции. Это обеспечивает достаточно быстрый доступ к компонентам вектора. Даже в языках типа Pascal, где каждый аргумент K может быть переменным, K нужно вычислить только один раз, когда вектору отводится определенное место в памяти. Следовательно, доступ к элементам массива эффек-

тивен даже в языке Pascal. Поскольку размер каждого компонента E известен уже при трансляции, то, если значение индекса также известно при трансляции (например, A[2]), вся формула доступа может быть вычислена во время компиляции. Обратите внимание на то, что в массивах типа char языка C E = 1; поскольку всегда выполняется равенство LB = 0, эквивалентная формула доступа в C становится чрезвычайно простой:

```
l-значение(A[I]) = α + I
```

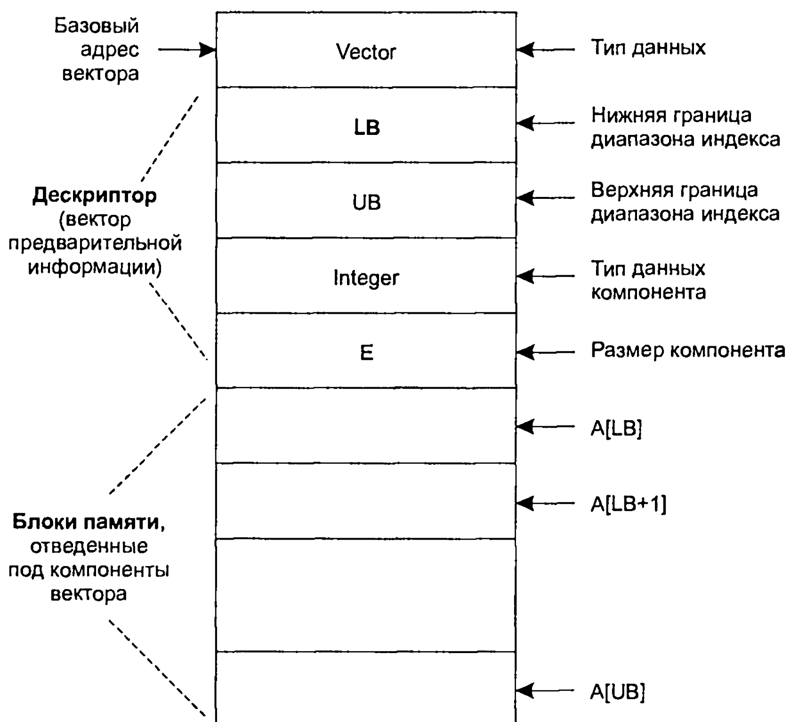


Рис. 6.3. Представление вектора A с полным дескриптором

Виртуальный начальный адрес. Теперь вычислим адрес элемента с индексом 0 нашего вектора. Формула доступа дает следующий результат:

```
l-значение(A[0]) = (α - LB × E) + (0 × E) = (α - LB × E) = K
```

Таким образом, K представляет собой адрес блока памяти, который занимает элемент вектора с индексом 0, *если он существует.* Поскольку элемент с нулевым индексом может и не существовать (потому что нижняя граница диапазона индексов может быть больше нуля), то этот адрес называется виртуальным начальным адресом (virtual origin, VO). Таким образом, мы получаем следующий алгоритм для построения векторов и генерации формулы доступа.

1. *При выделении памяти для представления вектора* отводится место в памяти для N компонентов вектора, каждый размером E, и для дескриптора размером D, то есть всего D + N × E ячеек памяти. Пусть α обозначает адрес расположения первого компонента вектора.

2. Вычисляем виртуальный начальный адрес $VO = \alpha - LB \times E$.
3. *При доступе к компоненту вектора l*-значение любого компонента A[I] вычисляется как

$$l\text{-значение}(A[I]) = VO + I \times E$$

Предполагается, что значение индекса I в формуле заведомо является допустимым для данного массива A. Для проверки возможных ошибок, связанных с выходом значения индекса за пределы допустимого диапазона, следует до вычисления формулы доступа проверить выполнение двойного неравенства $LB \leq I \leq UB$ (где LB и UB — соответственно нижняя и верхняя границы диапазона), а значения LB и UB должны присутствовать в дескрипторе во время выполнения программы. Вообще говоря, чтобы точно указать наиболее эффективный способ вычисления формулы доступа и проверки диапазона, нужно учесть конкретную конфигурацию аппаратной части компьютера и те операции, которые в нее встроены. Проверка диапазона именно этими средствами гарантирует, что даже если индекс I не находится внутри диапазона $LB \leq I \leq UB$, соответствующая область памяти никогда не может быть доступна как *l*-значение, которое не представляет правильный индекс.

Заметим, что в приведенной формуле доступа используется виртуальный начальный адрес для определения местоположения данного элемента массива. Если виртуальный начальный адрес хранится в дескрипторе массива, то сам массив не обязательно должен занимать область памяти, смежную с его дескриптором (рис. 6.4). Это обычный случай, когда дескрипторы для параметров-массивов могут быть переданы в какую-либо подпрограмму, а сам массив может фактически храниться где-то в другом месте. (Заметим, что типы данных для дескриптора и компонентов вектора на рис. 6.4 опущены. Для таких языков, как C или Pascal, эта информация получается во время трансляции, поэтому нет необходимости сохранять ее в дескрипторе времени выполнения.)

Упакованное и неупакованное представления в памяти. Предполагается, что размер каждого компонента в приведенной выше формуле доступа равен E и является целым числом адресуемых единиц памяти (слов или байтов). Например, если базовая адресуемая единица памяти — это слово, то предполагается, что каждый компонент занимает одно слово, или два, или более в зависимости от указанного в объявлении типа. Если тип компонента *логический* или *символьный*, то для представления этого компонента может потребоваться только небольшая часть слова. В таком случае в одно слово могут уместиться несколько компонентов. *Упакованное представление* — это такое представление, при котором компоненты вектора (или другой структуры) располагаются в памяти подряд, при этом каждый очередной компонент вовсе не должен оказываться в начале адресуемого слова (или байта). Упакованное представление позволяет сэкономить значительное количество ресурсов памяти. К сожалению, доступ к компонентам упакованного вектора обычно обходится гораздо дороже, поскольку в таком случае обычная формула доступа непригодна. Вместо нее приходится использовать ряд более сложных вычислений для доступа к тому слову или байту, в котором содержится искомый компонент.

Высокая стоимость доступа к упакованному вектору является причиной, по которой они обычно хранятся в *неупакованном* виде: каждый компонент распо-

лагается в начале отведенного ему байта или слова. Тогда упрощается доступ, зато возрастает объем занимаемой памяти. Но с течением времени машины, в которых адресуемой единицей памяти является байт, вытесняют те, в которых такими единицами служат слова; в связи с этим вопрос о выборе между описанными представлениями в настоящее время возникает довольно редко.

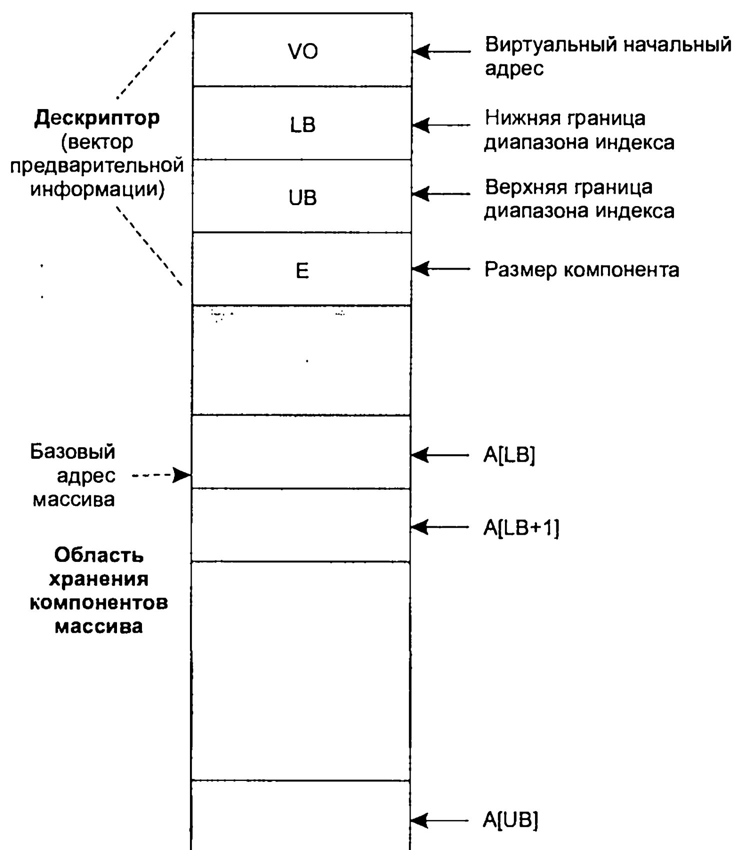


Рис. 6.4. Несмежные области хранения дескриптора и компонентов массива

Операции над векторами как отдельными объектами. Операции над векторами как отдельными объектами легко реализуются в случае последовательного представления вектора в памяти. Присвоение одного вектора другому, имеющему такие же атрибуты, реализовано как простое копирование содержимого блока памяти, представляющего первый вектор, в блок памяти, который представляет второй вектор (дескрипторы переписывать не нужно, потому что они одинаковые). Арифметические операции над векторами и такие специальные операции, как их скалярное произведение реализуются как циклы, в которых последовательно обрабатываются компоненты векторов.

Основная проблема реализации таких операций с векторами связана с ресурсами памяти, необходимыми для хранения результатов этих операций. В резуль-

тате сложения двух векторов, состоящих из ста элементов, получается третий вектор того же размера. В том случае, если полученный результат тут же не присваивается какому-либо существующему объекту, представленному своим *l*-значением, для хранения *r*-значения результата сложения требуется временно выделить для него память. Если в программе таких операций много, то общий объем памяти, необходимой для хранения промежуточных результатов, может оказаться достаточно большим и управление распределением этих дополнительных блоков памяти может значительно усложнить выполнение программы и увеличить ее стоимость.

Многомерные массивы

Вектор — это одномерный массив; матрица, составленная из строк и столбцов, является двухмерным массивом, трехмерный массив состоит из двухмерных плоскостей, содержащих строки и столбцы. Аналогично можно сконструировать массив любой размерности из массивов меньших размерностей. Представление в памяти векторов и формула доступа к отдельным элементам векторов легко могут быть обобщены на случай многомерных массивов.

Спецификация и синтаксис. Атрибуты многомерного массива отличаются от атрибутов вектора только тем, что требуется указывать диапазон изменения значений индекса для каждого измерения отдельно, как, например, в следующем объявлении языка Pascal:

```
B: array [1..10, -5..5] of real;
```

Выбор компонента требует указания одного индекса для каждого измерения (например, B[2,4] для двухмерного массива).

Реализация. Матрицу удобно реализовать, если рассматривать ее как вектор, состоящий из векторов; трехмерный массив можно рассматривать как вектор, компонентами которого являются матрицы и т. д. Заметим, что все эти компоненты должны, в свою очередь, состоять из одинакового количества элементов одного типа.

В некоторых случаях имеет значение, как мы рассматриваем матрицу — как строку, элементами которой являются столбцы, или как столбец, элементами которого являются строки (чаще всего это проявляется при передаче матрицы в качестве фактического параметра подпрограмме, написанной на другом языке). Более распространенным является взгляд на матрицу как на структуру, представляющую столбец строк, то есть вектор, каждый элемент которого также является вектором, представляющим один ряд исходной матрицы. Такое представление матрицы называется ее *развертыванием по строкам*. Вообще говоря, массив любой размерности организован развертыванием по срокам, причем сначала массив рассматривается как вектор по первому индексу, и элементами этого вектора являются массивы, размерность которых на единицу меньше размерности исходного массива. Затем каждый из этих массивов на единицу меньшей размерности (для каждого элемента вектора) снова рассматривается как вектор по второму индексу (относительно исходного массива), причем размерности получившихся его элементов-массивов снова уменьшаются на единицу и т. д. *Развертывание по столбцам* — это такое представление матрицы, когда она рассматривается как одна строка, элементами которой являются столбцы.

Представление многомерного массива в памяти следует из представления вектора. В случае, если матрица развертывается по строкам, сначала располагаются объекты данных, представляющие первую строку, затем следуют объекты данных, входящие во вторую строку и т. д. В результате получается единый блок данных, организованный последовательно, в котором поочередно расположены все компоненты массива. Дескриптор массива отличается от дескриптора вектора только тем, что в случае массива требуется указание верхней и нижней границ диапазонов изменения индексов каждого измерения. На рис. 6.5 схематически изображено представление матрицы в памяти.

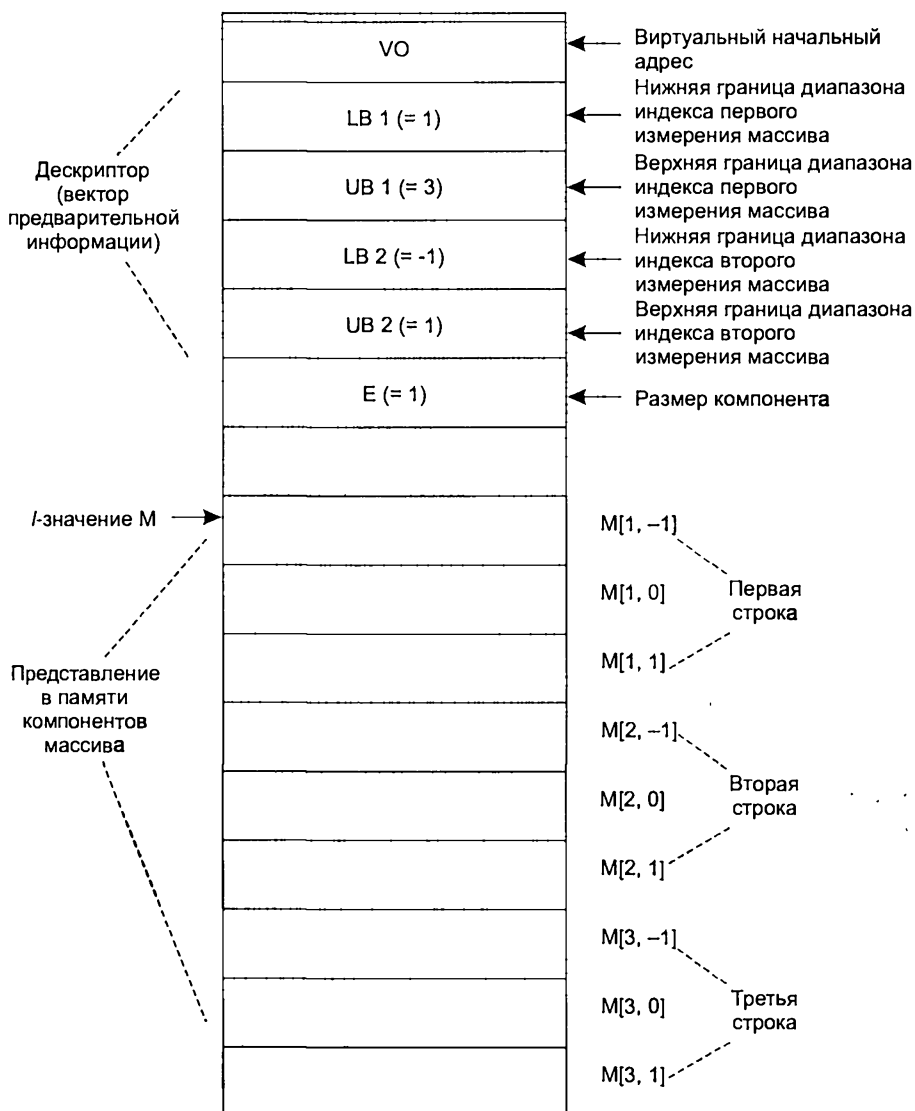


Рис. 6.5. Представление в памяти двухмерного массива

Операция индексации использует формулу доступа для вычисления смещения местоположения компонента от базового адреса в массиве, аналогичную той, которая определена для векторов: для получения *l*-значения компонента A[I,J] сначала нужно определить количество строк, которые следует пропустить, $(I - LB_1)$, и умножить это число на длину строки, что даст местоположение начала строки с индексом I, а затем найти в этой строке компонент под номером J аналогично тому, как это делается для вектора. Таким образом, если A — матрица с M строками и N столбцами и A представлена при помощи развертывания по строкам, то местоположение элемента A[I,J] определяется следующим образом:

$$l\text{-значение}(A[I,J]) = \alpha + (I - LB_1) \times S + (J - LB_2) \times E$$

где α — базовый адрес; S — длина строки $= (UB_2 - LB_2 + 1) \times E$; LB_1 — нижняя граница диапазона значений первого индекса; LB_2, UB_2 — нижняя и верхняя границы диапазона значений второго индекса.

Приведенную формулу можно упростить, выделяя постоянные члены при помощи следующих замен:

$$S = (UB_2 - LB_2 + 1) \times E$$
$$VO = \alpha - LB_1 \times S - LB_2 \times E$$
$$l\text{-значение}(A[I,J]) = VO + I \times S + J \times E$$

Как и в случае вектора, через E обозначен размер каждого компонента, а через S — длина каждой строки матрицы. Это позволяет легко вычислить VO.

Обратите внимание на то, что VO, S, α и E определяются еще при создании массива и, следовательно, их нужно вычислить только один раз и сохранить. Как и в случае вектора, VO является виртуальным начальным адресом и представляет собой местоположение в памяти компонента A[0, 0], если таковой существует в данном массиве. Тогда в самом худшем случае для доступа к компоненту массива придется провести следующее вычисление:

$$l\text{-значение}(A[I,J]) = VO + I \times S + J \times E$$

Массивы более высоких размерностей. Обобщение приведенных формул на случай массивов более высоких размерностей достаточно очевидно. Можно использовать следующий общий алгоритм для размещения дескрипторов при создании массива и для доступа к его элементам.

Предположим, у нас имеется массив $A[L_1 : U_1, \ldots, L_n : U_n]$ размерности n, где L_i — нижние границы, U_i — верхние границы i-го измерения, а e — размер каждого элемента массива. Предположим, что начало массива расположено по адресу α.

- *Вычисление множителей.* Каждый множитель m_i вычисляется следующим образом:

  $m_n = e$

  Для каждого i от $n - 1$ до 1 вычисляем $m_i = (U_{i+1} - L_{i+1} + 1) \times m_{i+1}$.

- *Вычисление виртуального начального адреса:*

  $VO = \alpha - \Sigma^n_{i=1} (L_i \times m_i)$

  Множители m_i и виртуальный начальный адрес VO могут храниться в дескрипторе массива во время выполнения программы.

✦ *Адрес элемента массива.* Адрес элемента A[s_1,..., s_n] определяется по следующей формуле:

$VO + \Sigma^n_{i=1}(s_i \times m_i)$.

Сечения

Спецификация. *Сечение* — это подструктура массива, которая сама является массивом. На рис. 6.6 приведены некоторые примеры сечений: на рис. 6.6, *а* представлено сечение в виде второго столбца матрицы, состоящей из трех столбцов; на рис. 6.6, *б* сечение представлено третьей строкой матрицы, состоящей из четырех строк, а на рис. 6.6, *в* сечением является третья плоскость трехмерного массива.

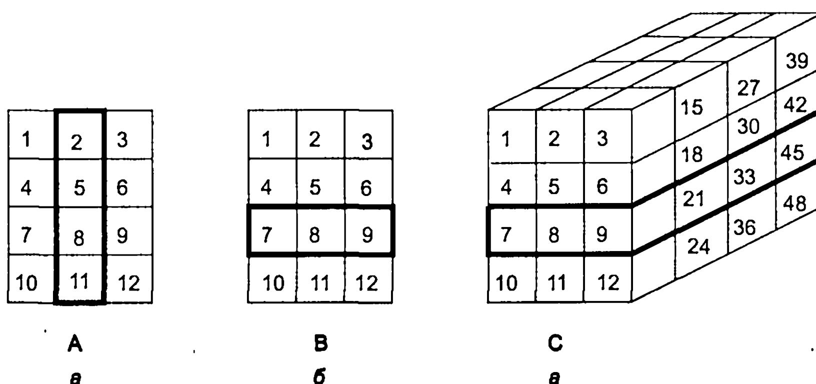


Рис. 6.6. Сечения массивов: *а* — одномерное сечение (столбец); *б* — одномерное сечение (строка); *в* — многомерное сечение (плоскость)

Одним из первых языков, в которых были реализованы сечения, является PL/I. Если массив объявлен как A(4, 3), то сечение, изображенное на рис. 6.6, *а*, обозначается через A(*, 2), где символ * означает, что первый индекс (строка) меняется от 1 до 4. Аналогично два других сечения этого рисунка можно задать как B(3, *) и C(3, *, *). Сечения можно передавать подпрограммам в качестве аргументов.

В языке FORTRAN в подпрограммы разрешается передавать часть массива в качестве параметров. Поскольку в FORTRAN используется развертывание массивов по столбцам, то при передаче элемента A(1, 3) векторному параметру B устанавливается соответствие между A(1, 3) и B(1), A(2, 3) и B(2), A(3, 3) и B(3) и т. д. Это позволяет программистам разрабатывать эффективные матричные алгоритмы обработки большой матрицы по частям. Однако для того, чтобы воспользоваться этими алгоритмами, требуется знание внутренних механизмов представления матриц в языке FORTRAN.

Поскольку этот метод противоречит современной концепции разработки языка, в FORTRAN 90 реализовано понятие сечений. Сечения, изображенные на рис. 6.6, в FORTRAN 90 задаются как A(1 : 4,2), B(3,1 : 3) и C(3,1 : 3,1 : 4).

Реализация. Использование дескрипторов позволяет эффективно реализовать сечение. Например, матрица A размером 3 × 4 описывается дескриптором.

VO	a - 4
LB 1	1
UB 1	4
Множитель 1	3
LB 2	1
UB 2	3
Множитель 2	1

В этом случае формула вычисления адреса элемента A[I, J], приведенная в конце предыдущего раздела, будет выглядеть следующим образом:

`l-значение(A[I, J]) = VO + I × 3 + J × 1`

Обратите внимание на то, что в этом случае множитель 2, который представляет собой размер объекта данных, также определяет расстояние между последовательными элементами массива. В данном случае элементы массива следуют непрерывно один за другим. Однако это не обязательно; сечение обладает таким свойством, что все его элементы расположены на одинаковом расстоянии друг от друга, но при размещении массива в памяти они не следуют непрерывно друг за другом. Следовательно, дескриптор сечения A(*, 2) (см. рис. 6.6), учитывая дескриптор всего массива A, можно записать следующим образом.

VO	α - 2
LB 1	1
UB 1	4
Множитель 1	3

Этот дескриптор описывает вектор, состоящий из четырех элементов, виртуальная начальная позиция которого смещена на две позиции памяти (VO + 2 × 1) относительно виртуальной начальной позиции VO массива A и элементы которого разнесены с промежутком, равным трем позициям памяти. Аналогично мы можем представить дескрипторы для сечений B(3,*) и C(3,*,*) (см. рис. 6.6), причем для второго сечения дескриптор будет двухмерным.

Ассоциативные массивы

Характерной чертой тех массивов, которые мы изучили к настоящему моменту, был метод доступа к отдельным их элементам с помощью перечисляемого типа данных под названием *индекс*. Элементы массива были упорядочены в соответствии с этим индексом, и доступ к каждому элементу осуществлялся посредством выбора соответствующего значения индекса.

В некоторых приложениях желательно получать доступ к элементам массива по имени без предварительного упорядочения соответствующих индексов. Например, список оценок учеников некоторого класса можно организовать в виде двух массивов, в одном из которых содержатся имена студентов, name[i], а во втором — полученные ими оценки, grade[i]. В данном случае подразумевается упорядочение по целочисленному индексу i.

Альтернативный метод заключается в том, чтобы использовать в качестве индекса имя. Поскольку последовательность имен (то есть строк символов) фактически неограниченна, то не существует какого-либо реального неявного перечисления. Вместо этого множество имен используется как множество перечисления. При добавлении нового имени это перечисление увеличивается. Такой массив называется *ассоциативным массивом*. Ассоциативные массивы в SNOBOL4 называются таблицами и объявляются с помощью ключевого слова table, они также играют важную роль в таких языках обработки процессов, как Perl[1].

В Perl ассоциативный массив создается при помощи специальной операции, которая обозначается символом %. Так, следующий оператор:

```
%ClassList = ("Michelle", 'A', "Doris", 'B', "Michael", 'D');
```

создает ассоциативный массив из трех элементов. Первый член каждой пары является ключом, а второй — значением. Доступ к ассоциативному массиву %ClassList продемонстрирован в следующем примере:

```
$ClassList{'Michelle'};          # значение равно A
@y=%ClassList;                   # Массив y объявляется массивом
                                 # скаляров из 6 элементов
for ($i=0; $i<6; $i++){print "I=, $i, $y[$i]\n"};
```

Печать из цикла for будет выглядеть так:

```
I=, 0, Doris
I=, 1, B
I=, 2, Michael
I=, 3, D
I=, 4, Michelle
I=, 5, A
```

Этот пример помогает понять реализацию ассоциативных массивов. Хотя исходный порядок ключей элементов массива был (Michelle, Doris, Michael), при хранении в памяти массив упорядочивается по алфавиту (точнее, в соответствии с алфавитным порядком ключей массива). Это обеспечивает возможность эффективного поиска ключа нужного элемента массива на основе алгоритма двоичного поиска[2].

6.1.6. Записи

Структура данных, состоящая из фиксированного количества компонентов различных типов, обычно называется *записью*.

Спецификация и синтаксис. Записи и векторы являются различными формами линейной структуры данных фиксированных размеров, но записи отличаются от векторов в двух отношениях:

---

[1] Ассоциативные массивы можно также найти в языке PHP, достаточно широко применяемом для создания серверных сценариев в web-приложениях. — *Примеч. науч. ред.*

[2] Этот абзац не соответствует действительности. Ассоциативные массивы Perl вообще никак не упорядочиваются при хранении, более того, нельзя даже предугадать их порядок, так как для вычисления адреса области памяти элемента используется хэш-функция. Чтобы получить упорядоченный набор ключей, следует использовать функцию сортировки: @y=sort(keys(%ClassList));. — *Примеч. науч. ред.*

1) компоненты записей могут быть разнородными, то есть объектами данных разных типов, в отличие от однородных элементов векторов;
2) компоненты записей обозначаются *символическими именами* (*идентификаторами*) в отличие от индексов, нумерующих элементы векторов.

Достаточно типичным примером синтаксиса, используемого для объявления записей, является конструкция struct, используемая в языке С:

```
struct EmployeeType
{int ID;
int Age;
float SALARY;
char Dept;
} Employee;
```

Это объявление определяет запись типа EmployeeType, состоящую из четырех компонентов, два из которых целочисленного типа: один — вещественного и один — символьного с именами ID, Age, SALARY и Dept соответственно. Employee объявлена как переменная типа EmployeeType (в следующих объявлениях переменных этого типа не требуется описывать внутреннюю структуру записи EmployeeType). Для того чтобы выбрать компонент записи, в языке С используется следующая синтаксическая конструкция:

```
Employee.ID
Employee.SALARY
```

Атрибуты записи видны из приведенного выше объявления:

1) количество компонентов;
2) тип данных для каждого компонента;
3) имя для обозначения каждого компонента.

Компоненты записи часто называются *полями*, и соответственно имена компонентов являются *именами полей*. Записи иногда называются структурами (как в С).

Выбор компонентов является одной из основных операций над записями, например Employee.SALARY. Она соответствует выбору элементов из массива при помощи индексов, но с одним существенным отличием: индекс здесь всегда является буквальным именем компонента и никогда не может быть вычисляемым значением. Приведенный пример выбора третьего компонента записи, Employee.SALARY, соответствует выбору третьего компонента вектора, VECT[3], но для записей не существует аналогичной векторной операции выбора VECT[I], где I — вычисляемое значение.

Операции над записью как единым целым обычно немногочисленны. Чаще всего используется операция присваивания одной записи некоторой другой, имеющей такую же структуру, например:

```
struct EmployeeType INPUTREC;
...
Employee = INPUTREC;
```

где INPUTREC — запись, имеющая те же атрибуты, что и Employee. Соответствие имен компонентов различных записей позволяет применять присваивание в языках COBOL и PL/I. Например, в COBOL оператор

```
MOVE CORRESPONDING INPUTREC TO EMPLOYEE
```

присваивает каждый компонент записи INPUTREC соответствующему компоненту записи EMPLOYEE, где соответствующие компоненты должны иметь одинаковые имена и типы данных, но порядок их расположения в каждой записи может быть произвольным.

Реализация. Представление записи в памяти состоит из единого непрерывного блока памяти, в котором компоненты расположены последовательно (рис. 6.7). Для указания типов данных или других атрибутов отдельных компонентов могут потребоваться дескрипторы, но обычно для компонентов записи не требуется никаких дескрипторов во время выполнения программы.

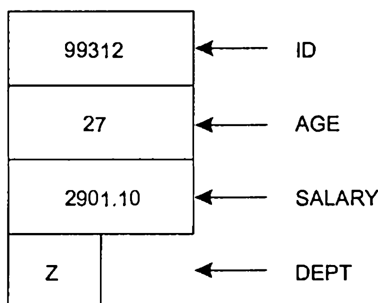


Рис. 6.7. Представление в памяти записи Employee

Поскольку индексы отдельных компонентов (имена полей) известны еще во время компиляции (их не надо вычислять во время выполнения программы), то реализация операции выбора достаточно проста. Объявление записи также позволяет определить размер каждого компонента и его позицию внутри блока памяти еще во время трансляции. В результате *смещение* каждого компонента также можно вычислить во время трансляции. Основная формула доступа, которая используется для вычисления местоположения I-го компонента, выглядит следующим образом:

$$l\text{-значение}(R.I) = \alpha + \sum_{j=1}^{I-1}(\text{размер } R.j)$$

где α — базовый адрес блока памяти, содержащего запись R, а R.j — компонент под номером j. Суммирование здесь необходимо из-за возможного различия размеров каждого компонента. Но эту сумму всегда можно вычислить во время трансляции и тем самым получить значение сдвига K_I для I-го компонента, так что во время выполнения потребуется добавить только базовый адрес основного блока памяти:

$$l\text{-значение}(R.I) = \alpha + K_I$$

Отводимые для хранения некоторых типов данных области памяти должны начинаться с определенных адресов. Например, области памяти, выделяемые под целые числа, должны начинаться на границе слова. Это означает (в случае, если в данной машине используются байты как основная адресуемая единица памяти), что адрес области памяти, используемый для представления целого числа, должен быть кратным 4 (то есть двоичный адрес должен заканчиваться на 00). Следовательно, отдельные элементы записей могут располагаться не вплотную друг к другу. Например, для структуры языка С, определяемой оператором,

```
struct EmployeeDivision
     { char Division;
     int IdNumber; } Employee;
```

область памяти, отводимая под поле IdNumber, должна начинаться на границе слова, и поэтому три байта между полями Division и IdNumber никак не используются и заполняются незначащей информацией. Представление в памяти такой записи равносильно представлению в памяти записи, создаваемой следующим объявлением:

```
struct EmployeeDivision
      { char Division;
      char UnusedPadding[3];
      int IdNumber; } Employee;
```

Операция присваивания целой записи некоторой другой, обладающей такой же структурой, может быть реализована как простое копирование содержимого блока памяти, представляющего первую запись, в блок памяти, представляющий вторую запись. Более сложная операция MOVE CORRESPONDING может быть реализована как последовательность операций присваивания отдельных компонентов одной записи компонентам другой записи.

Записи и массивы со структурированными компонентами

В тех языках, в которых предусмотрены в качестве основных типов данных и записи, и массивы, обычно допускается использование компонентов этих двух типов совместно с компонентами элементарных типов (и, как правило, других структурированных типов, таких как строки символов). Например, может оказаться полезным такой объект, как вектор, компонентами которого являются записи. Например, следующее объявление языка C:

```
struct EmployeeType
      {int ID;
      int Age;
      float SALARY;
      char Dept;
      } Employee[500];
```

объявляет массив, состоящий из пятисот компонентов, каждый из которых является записью типа EmployeeType. Компонент такой сложной структуры обычно выбирается при помощи последовательности операций выбора, причем сначала выбирается компонент вектора, а затем — компонент записи, например Employee[3].SALARY.

Запись также может состоять из компонентов, которые являются массивами или другими записями. В результате этого можно создавать записи, имеющие иерархическую структуру, на верхнем уровне которой располагаются компоненты, сами являющиеся массивами или записями. Компонентами второго уровня иерархии также могут быть записи или массивы. В языках COBOL и PL/I эта иерархическая организация выражена синтаксически посредством присваивания *номеров уровней* для указания каждого нового уровня компонентов. Типичным в этом отношении является объявление PL/I:

```
1 Employee,
   2 Name,
      3 Last CHARACTER(10),
      3 First CHARACTER(15),
      3 Middle CHARACTER(1),
   2 Age FIXED(2),
   2 Address,
```

```
3 Street.
   4 Number FIXED(5).
   4 St-Name CHARACTER(20).
3 City CHARACTER(15).
3 State CHARACTER(10).
3 Zip FIXED(5):
```

Синтаксис этого объявления похож на краткое содержание книги с заголовками, подзаголовками и т. д. Структура данных, определяемая этим объявлением, состоит из одной записи Employee, компоненты второго уровня которой представлены именами Name, Age и Address. Age — это компонент элементарного типа (целое число), а компоненты Name и Address являются записями, компоненты которых расположены уже на третьем уровне иерархии. Компонент Street записи Address также является записью, причем ее компоненты являются компонентами уже четвертого уровня иерархии.

Реализация. Представление в памяти векторов и записей, компонентами которых являются другие векторы и записи, является простым расширением представления простых векторов и записей. В разделе 6.1.5 мы рассматривали векторы, состоящие из векторов. Вектор, состоящий из записей, представлен в памяти так же, как и простой вектор, составленный из целых чисел или компонентов другого элементарного типа, с тем лишь отличием, что блок памяти, представляющий компонент вектора в блоке большего размера, отведенном для самого вектора, является блоком памяти, выделенным для хранения записи. Таким образом, вектор записей представлен в памяти примерно так же, как вектор, состоящий из векторов (см. рис. 6.5), но каждая строка в нем заменена на представление записи в памяти.

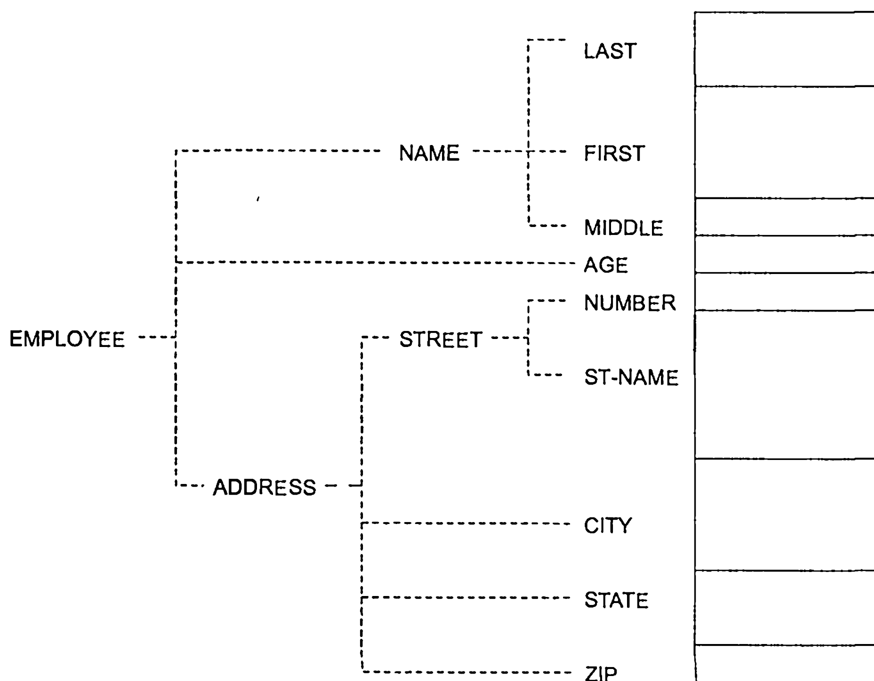


Рис. 6.8. Представление многоуровневой записи языка PL/I

Аналогично запись, компонентами которой являются записи (или векторы), сохраняет свою последовательную структуру представления в памяти, но подблоки, отведенные под отдельные компоненты, могут являться представлениями целой записи. Рис. 6.8 иллюстрирует представление в памяти структуры данных PL/I, объявление которой приведено выше. Выбор отдельных компонентов происходит последовательно — сначала определяется базовый адрес всей структуры в целом, затем вычисляется сдвиг для компонента первого уровня иерархии, после чего вычисляется смещение относительно этого базового адреса для нахождения местоположения компонента второго уровня и т. д.

Вариантные записи

Записи часто используются для представления похожих объектов, которые могут содержать различающиеся данные. Например, зарплата служащему может начисляться за отработанные часы (почасовая) или за работу в течение месяца (оклад). Может оказаться полезным иметь один тип записи, который, впрочем, имеет несколько вариантов. В такой записи имеются один или несколько компонентов, общих для всех вариантов, а также несколько других компонентов, чьи имена и типы данных уникальны для каждого варианта записи. Например, в записи, содержащей информацию о зарплате служащих, могут быть два варианта — для тех служащих, которые имеют ежемесячную фиксированную зарплату (оклад), и для тех, кому установлена почасовая оплата. Объявление языка Pascal

```
type PayType = (Salaried, Hourly);
var Employee: record
                                    ID: integer;
                                    Dept: array [1..3] of char;
                                    Age: integer;
                                    case PayClass: PayType of
                                      Salaried: (MonthlyRate: real;
                                                 StartDate: integer);
                                      Hourly: (HourRate: real;
                                               Reg: integer;
                                               Overtime: integer)
                                    end
```

определяет такую вариантную запись. В этой записи всегда присутствуют компоненты ID, Dept, Age и PayClass. Если значение компонента PayClass равно Salaried, то в записи также присутствуют компоненты MonthlyRate и StartDate, в то время как если PayClass = Hourly, то в записи будут содержаться компоненты HourRate, Reg и Overtime. Компонент PayClass называется *тегом* (Pascal) или *дискриминантом* (Ada), поскольку он служит для указания варианта записи, который должен использоваться в данном месте программы.

Операция выбора компонента вариантной записи аналогична операции выбора компонента обычной записи. Например, Employee.MonthlyRate и Employee.Reg выбирают компоненты из определенных ранее вариантов записи Employee. Для обычных записей каждый компонент существует в течение всего времени жизни записи, а в случае вариантной записи компонент может просуществовать какое-то время (пока тег имеет соответствующее значение), затем прекратить свое существование (если тег меняет значение) и вновь появиться впоследствии при изменении значения тега на первоначальное. Таким образом, выбор компонента Employee.Reg может

означать попытку выбрать не существующий в данный момент компонент. Проблема выбора не существующего в программе компонента вариантной записи аналогична ошибке, связанной с указанием индекса вне диапазона его значений для массивов. Эту проблему мы обсуждали в предыдущем разделе, и методы ее решения для вариантных записей похожи на те, что используются вообще для всех структурированных данных.

1. *Динамическая проверка.* Во время выполнения программы до того, как осуществлять выбор нужного компонента, можно проверить значения компонента-тега. Это позволит убедиться в том, что компонент в данный момент существует. Если значение тега правильное, то нужный компонент доступен; в противном случае это является ошибкой времени выполнения программы, и тогда инициируется специальный способ ее обработки, во многом схожий с проверкой принадлежности индекса массива диапазону его значений.
2. *Отсутствие проверки.* Язык программирования может быть разработан таким образом, что он допускает определение вариантной записи без явного указания тега-компонента, содержимое которого можно было бы проверить во время выполнения программы, поэтому всегда предполагается, что выбор компонента такой записи правилен. Из-за способа реализации вариантных записей, которую мы обсудим чуть позже, подобный выбор всегда возможен. Однако если компонент не существует, то существующие в данный момент значения вариантных компонентов могут быть нечаянно переписаны или использованы не по назначению. Языки PL/I, COBOL и Pascal позволяют создавать формы вариантных записей без определения полей тегов, а объявление объединений union языка C вообще не допускает создания тега. Реализация этих форм не позволяет осуществлять проверку наличия искомых компонентов.

Вариантные записи также часто называются *объединением*, поскольку каждый вариант записи можно рассматривать как отдельный класс объектов данных типа запись, а общий для них тип записи получается затем как объединение этих множеств объектов данных. Если в типе не определено поле тега (как в случае типа union в языке C), то это тип со *свободным объединением*; если же поле указано, то это тип с *дифференцированным* (*различаемым*) *объединением*. Термин *дифференцируемый* указывает на то обстоятельство, что возможно (с помощью проверки поля тега) определить класс вариантов, к которому принадлежит каждый объект данных из их общего типа.

Реализация. Реализовать вариантную запись проще, чем правильно ее использовать. Во время трансляции определяется количество памяти, необходимое для хранения всех компонентов каждого варианта, и отводится такое количество памяти, которое необходимо для хранения *максимального* возможного варианта (рис. 6.9). Внутри этого блока памяти каждый вариант описывает различную компоновку блоков в терминах количества и типов компонентов. Поскольку размер выделенного блока достаточен, чтобы вместить самый большой вариант записи, то, естественно, во время выполнения программы в нем сможет разместиться любой вариант записи, но часть выделенной памяти не будет использоваться вариан-

тами меньших размеров. Компоновки различных вариантов определяются во время трансляции и используются для вычисления смещения при выборе компонентов. Во время выполнения для вариантной записи не требуется никакого специального дескриптора, поскольку тег-компонент рассматривается просто как другой компонент записи.

Рис. 6.9. Представление в памяти вариантных записей

Выбор компонента определенного варианта записи полностью аналогичен выбору компонента обычной записи, когда отсутствует всякая проверка типа. Во время трансляции вычисляется смещение искомого компонента относительно базового адреса, выделенного под всю запись блока памяти; во время выполнения программы это смещение добавляется к базовому адресу для определения местоположения компонента в памяти. Если компонент в данный момент выполнения программы не существует, то полученный адрес соответствует тому месту, где он *находился бы*, если бы существовал, и эта область памяти будет содержать значение (или часть значения, или несколько значений), представляющее значение компонента текущего варианта записи. Присваивание какого-либо значения не существующему в данный момент компоненту (в отсутствие проверки) изменяет содержимое области памяти, в которой он должен был бы находиться. Если в данный момент эта область используется как часть компонента текущего варианта записи, тогда могут произойти непредсказуемые изменения значения этого компонента (с потенциально катастрофическими для программы последствиями).

Если при выборе компонентов вариантной записи применяется динамическая проверка типов, то во время выполнения программы точно так же вычисляется значение суммы базового адреса и смещения для определения местоположения компонента, но в первую очередь проверяется значение поля тега, чтобы убедиться, что в данный момент в памяти существует требуемый текущий вариант записи.

6.1.7. Списки

Структура данных, состоящая из упорядоченной последовательности структур данных, обычно называется *списком*.

Спецификация и синтаксис. Списки похожи на векторы в том отношении, что они представляют собой упорядоченную последовательность объектов данных. Это означает, что можно определить первый элемент списка (который обычно называется *головой* списка), второй элемент и т. д. Однако между списками и векторами есть некоторые существенные различия.

1. Списки редко имеют фиксированную длину. Они часто используются для представления произвольных структур данных, и обычно их длина увеличивается и уменьшается в процессе выполнения программы.
2. Списки редко бывают однородными. Тип данных каждого элемента списка может отличаться от типа его соседей.
3. В тех языках, в которых используются списки, типы данных элементов списка объявляются неявным образом без явного задания атрибутов элементов списков.

Синтаксис языка LISP представляет типичную списковую структуру:

```
(FunctionName Data1 Data2 … Datan)
```

Эта конструкция означает, что функция FunctionName применяется последовательно к объектам $Data_1$, $Data_2$, ..., $Data_n$.

Большинство операций языка LISP в качестве аргументов получают список, и результатом их выполнения также является список значений. Например, операция cons в качестве аргументов использует два списка и возвращает список, являющийся последовательным объединением двух исходных, то есть первый список добавляется в начало второго:

```
(cons '(a b c) '(d e f)) = ((a b c) d e f)
```

Этот пример показывает, что результатом операции cons является список из четырех элементов, первым из которых служит список (a b c). Четыре элемента полученного списка являются элементами не одного и того же типа: первый элемент сам является списком, а остальные являются элементарными объектами данных, или *атомами*.

Этот пример также иллюстрирует одно характерное свойство языка LISP. Сначала вычисляются все аргументы функции. Если бы это выражение было записано без апострофов:

```
(cons (a b c) (d e f))
```

то LISP сначала попытался бы вычислить функцию a с аргументами b и c, а затем — функцию d с аргументами e и f. Вероятно, это привело бы к ошибке. Функция (quote x), или, проще, 'x, просто возвращает буквальное значение (*l*-значение) своего аргумента, позволяя тем самым избежать ненужных в данном случае вычислений.

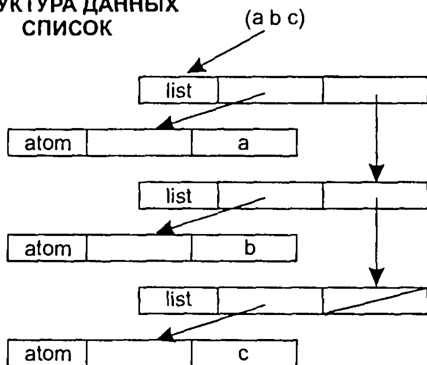
В языке ML также определены списки. Их синтаксис таков: [a, b, c]. В отличие от LISP в ML списки должны быть однородны, поэтому допускаются списки, состоящие из целых чисел (например, [1, 2, 3]), и списки, состоящие из строк (например, ["abc", "def", "ghi"]).

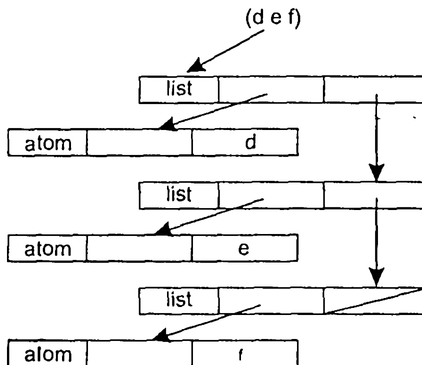
Реализация. Динамическая природа большинства реализаций списков и тот факт, что элементы списка редко бывают однородными, означают, что регулярное управление распределением памяти, полезное для реализации векторов и массивов, не будет работать для списков. В таких случаях, как правило, используется организация управлением памятью на основе *связанных списков*. Элемент списка является простейшим элементом и обычно представляет собой объект данных фиксированного размера. В языке LISP для представления списка обычно необходимо три информационных поля: поле типа и два указателя списков. Если в поле типа задан атом, то остальные два поля являются дескрипторами, описывающими этот атом. Если в поле типа задан список, то первый указатель является *головой* (head) списка (его первым элементом), а второй указатель является *хвостом* (tail) списка (его последующими элементами). На рис. 6.10 изображена структура представления памяти для предыдущего примера:

```
(cons '(a b c) '(d e f)) = ((a b c) d e f)
```

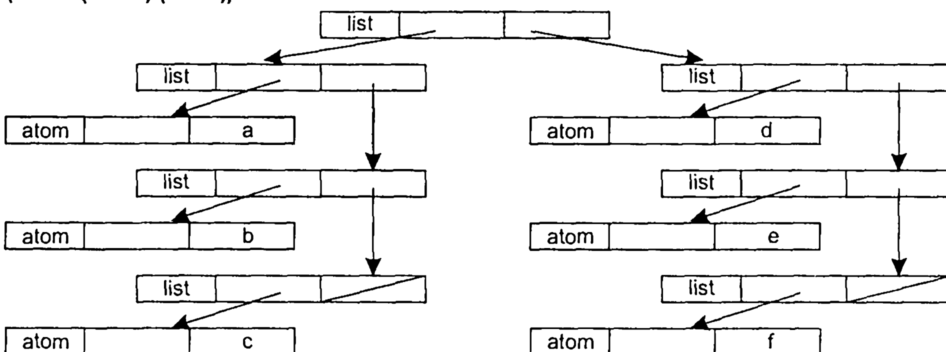
Тип	Поле головы списка	Поле хвоста списка

СТРУКТУРА ДАННЫХ СПИСОК





(CONS (A B C) (D E F))



СТРУКТУРА ДАННЫХ, ПОЛУЧЕННАЯ В РЕЗУЛЬТАТЕ ОПЕРАЦИИ CONS

Рис. 6.10. Представление списков в памяти

Сходство между средней и нижней структурами на этом рисунке подтверждает эффективность выбранной реализации списков.

1. cons. Операция cons, или join, реализуется так: создается новый узел списка и первый аргумент операции cons помещается в его головное поле, а второй ее аргумент — в хвостовое поле.
2. *Голова. Голова* списка — это содержимое (*r*-значение) головного поля элемента списка.
3. *Хвост. Хвост списка* — это содержимое хвостового поля элемента списка.

Язык LISP был реализован в начале 60-х гг. на компьютере IBM 704. Это была машина с 36-битовым словом, на которой два 18-битовых адреса могли храниться в одном 36-битовом регистре. Старшие 18 битов назывались *адресным регистром*, а младшие 18 битов — *декрементным регистром*. Следовательно, операция head (голова) представляла «содержимое адресного регистра» (contents of the address register, CAR), а операция tail (хвост) — «содержимое декрементного регистра» (contents of the decrement register, CDR). Эти аббревиатуры (CAR и CDR) настолько укоренились в фольклоре, связанном с языком LISP, что блюстители чистоты этого языка никогда не назовут подобные операции head и tail.

Списки являются базовыми объектами данных в таких языках, как ML, LISP и Prolog, но не входят в определение обычных компилируемых языков типа C, Pascal или Ada. Динамическое управление памятью, необходимое для представления списков, противоречит принципу максимальной эффективности регулярных структур управления памятью компилируемых языков. Тем не менее в программе, написанной на каком-либо компилируемом языке, все же могут использоваться списки, но только как определяемый программистом тип данных.

Различные виды списков

В некоторых языках определены различные вариации обычных списков.

Стеки и очереди. *Стек* — это такой список, в котором операции выбора, вставки и удаления компонента ограничены только одним его концом. *Очередь* — это такой список, в котором выбор и удаление компонента осуществляются с одного его конца, а добавление — с другого. Для очередей и стеков используется как последовательное, так и связанное представление.

Деревья. Список, компонентами которого могут быть как списки, так и элементарные объекты данных, называется деревом, при условии, что каждый список является компонентом не более чем одного другого списка. Большая часть примеров, иллюстрирующих LISP в этой книге (см. рис. 6.10), в действительности являются деревьями.

Ориентированные графы. Структура данных, в которой компоненты могут быть связаны друг с другом произвольным образом (то есть не обязательно последовательным способом), называется ориентированным графом.

Списки свойств. Запись с переменным количеством компонентов обычно называется списком свойств, если количество компонентов может варьироваться без каких-либо ограничений. (Структура вариантных записей в Pascal может изменяться только в рамках строго определенного набора альтернатив.) В списке свойств должны присутствовать и имена компонентов (имена полей), и их зна-

чения. Каждое имя называется именем свойства; значение соответствующего поля называется значением свойства. Общим представлением списков свойств являются обычные связанные списки, в которых имена свойств и их значения чередуются. Для того чтобы выбрать конкретное значение свойства (например, значение свойства Age (возраст)), осуществляется поиск только по именам свойств в списке, пока не будет найдено нужное имя. Следующий компонент списка — это значение этого свойства. Когда в список свойств вставляется новое свойство, фактически добавляются два объекта: имя свойства и его значение. В различных языках программирования списки свойств называются по-разному; в LISP они называются также таблицами (и для их хранения в памяти иногда используется смешанное последовательно-связанное представление). Также используются термины «список атрибутов-значений» (attribute-value list) и «список описаний» (description list).

Объекты данных переменных размеров естественно использовать в тех задачах, в которых количество данных заранее неизвестно. Структуры данных переменного размера позволяют выделять память по мере необходимости во время выполнения программы.

В языках программирования существуют два различных по существу подхода к использованию подобных типов данных. В некоторых языках, например в ML и LISP, такие типы данных, как список, список свойств, очередь и стек предоставляются непосредственно языком, то есть встроены в него. Реализация языка включает в себя скрытую систему управления памятью, которая автоматически распределяет память для этих структур и восстанавливает ее для последующего использования в случае их уничтожения в программе. В других языках, например в C, Pascal и Ada, предоставляется тип данных указатель вместе со средством явного динамического выделения памяти программистом, с помощью которых программист и строит связанные структуры, как описано в разделе 5.3.2.

Стеки, очереди, деревья и другие типы объектов данных переменного размера очень важны во многих фазах реализации самого языка. Например, стек, создаваемый во время выполнения программы, является одним из центральных объектов данных, определяемых системой, в реализациях большинства языков программирования (см. главу 9), деревья часто используются для представления таблиц символов в компиляторе (глава 3), а очереди часто используются для планирования и синхронизации выполнения параллельных подпрограмм.

6.1.8. Множества

Множество — это объект данных, содержащий *неупорядоченный* набор *различных* значений. В отличие от множества список является упорядоченным набором значений, некоторые из них могут повторяться. Основные операции над множествами таковы.

1. *Определение принадлежности к множеству.* Эта операция дает ответ на вопрос: является ли объект X элементом множества S (то есть верно ли, что $X \in S$)?
2. *Вставка и уничтожение отдельных значений.* Первая операция включает X в множество S, если там еще нет такого элемента. Вторая операция удаляет X из S, если X является элементом множества.

3. *Объединение, пересечение и разность множеств.* Пусть имеются два множества S_1 и S_2. Каждая из этих операций создает новое множество S_3, которое либо содержит элементы обоих множеств S_1 и S_2 (в случае *объединения*), либо содержит только те элементы, которые принадлежат и множеству S_1, и множеству S_2 (в случае *пересечения*), либо содержит только те элементы S_1, которых нет в S_2 (в случае *разности* множеств S_1 и S_2).

Заметим, что к компонентам множества невозможно осуществить доступ при помощи индексов или по их взаимному расположению.

Реализация. В языках программирования термин *множество* иногда применяется к структуре данных, которая представляет собой *упорядоченное множество*. Упорядоченное множество фактически является списком, из которого удалены повторяющиеся элементы. Такое множество не требует дополнительного рассмотрения. Неупорядоченное множество допускает два специальных способа представления, которые заслуживают отдельного обсуждения.

Представление множеств битовыми строками. Представление с помощью битовых строк уместно в том случае, если заранее известно, что используемый универсум (универсальное множество, из которого берутся значения, которые могут появиться в объектах данных множества) невелик по размерам. Предположим, что универсум состоит из N элементов. Расположим эти элементы некоторым произвольным образом: $e_1, e_2, ..., e_N$. Множество элементов, выбранных из этого универсума, можно представить битовой строкой длины N, в которой бит под номером i равен 1, если e_i входит в множество, и 0 в противном случае. Такая битовая строка представляет *характеристическую функцию* множества. В этом представлении операция вставки элемента в множество сводится к замене соответствующего бита в строке на 1, операция удаления — к замене соответствующего бита на 0, а определение принадлежности элемента области множеству сводится к проверке значения соответствующего бита. Операции объединения, пересечения и вычитания над множествами могут быть представлены при помощи поразрядных булевых операций над битовыми строками, которые обычно встроены в аппаратную часть компьютера: поразрядное логическое ИЛИ, примененное к двум битовым строкам, соответствует объединению, поразрядное логическое И соответствует пересечению, а операцию вычитания можно представить как поразрядное логическое И, примененное к первой битовой строке и инвертированной второй.

Если в компьютере предусмотрена аппаратная поддержка операций над битовыми строками, такой способ представления множеств становится особенно эффективным. Однако аппаратная поддержка таких операций над битовыми строками обычно реализована для битовых строк некоторой определенной длины (например, не более длины слова основной памяти). В случае более длинных строк приходится применять программное моделирование для разбиения длинной строки на несколько коротких, которые уже могут обрабатываться аппаратными командами.

Хэш-кодирование. Распространенной альтернативой представлению с помощью битовых строк служит представление множеств, основанное на технологии *хэш-кодирования*, или *фрагментированной памяти* (scatter storage). Этот метод следует использовать в том случае, если используемый универсум достаточно велик (например, если множество содержит числа или строки символов). Он позволяет

эффективно реализовать операции определения принадлежности к множеству, вставки элементов и (с помощью некоторых методов) удаления элементов множества. В то же время такие операции, как объединение, пересечение и вычитание множеств приходится реализовывать в виде последовательности операций определения принадлежности элемента множеству, вставки и удаления элементов, поэтому они весьма малоэффективны. Методы хэш-кодирования могут быть эффективными только при условии использования большого объема памяти. Как правило, язык не обеспечивает доступность для пользователя такого представления типа данных *множество*, но в реализации языка этот способ используется для представления некоторых данных, определяемых системой и используемых во время трансляции или выполнения программы. Например, большинство реализаций языка LISP используют этот способ для представления множества, называемого *списком объектов*, которое состоит из имен всех атомов, используемых во время выполнения конкретной программы. Почти все компиляторы используют хэш-кодирование для поиска имен в таблице символов (см. главу 3).

В случае векторов каждый компонент однозначно определяется своим индексом, и его адрес может быть легко получен с помощью простой формулы доступа. Целью хэш-кодирования является дублирование этой возможности, чтобы организовать эффективный доступ к компонентам множества. Но здесь возникает проблема, связанная с тем, что потенциальное множество допустимых имен огромно по сравнению с доступной памятью. Если мы выделим некоторый блок памяти, называемый хэш-таблицей, по меньшей мере в два раза превышающий тот, который мы предполагаем реально использовать, то хэш-кодирование может быть весьма эффективным (например, если в множестве будет 1000 элементов, то следует отвести место под 2000 элементов). Вместо того чтобы хранить элементы множества последовательно внутри выделенного блока памяти, они размещаются в нем произвольным образом. Хитрость заключается в способе размещения каждого нового элемента в выделенном блоке памяти таким образом, чтобы позже его присутствие или отсутствие можно было определить без организации поиска по блоку.

Рассмотрим, как это может быть сделано. Допустим, нам требуется добавить в множество S, представленное блоком памяти M_S, новый элемент x, представленный битовой строкой B_x. Сначала нам нужно определить, не содержится ли уже в этом множестве S элемент x; если не содержится, тогда нужно его добавить. Мы отводим некоторое место в блоке M_S для строки B_x, применяя так называемую *функцию хэширования* к битовой строке B_x. Эта функция хэширует (то есть разбивает на маленькие кусочки и затем перемешивает) строку B_x и в результате выдает некоторый хэш-адрес I_x. Этот хэш-адрес используется как индекс, указывающий на местоположение данной строки в блоке M_S. Если элемент x содержится в множестве S, то он должен находиться в том месте, на который указывает хэш-адрес I_x. Если элемента x в множестве нет, то в это место следует поместить строку B_x. Если впоследствии нужно узнать, содержится ли элемент x в множестве S, то это можно сделать посредством хэширования новой строки битов B_x, представляющей элемент x, и определения хэш-адреса I_x. По полученному адресу отыскивается нужная позиция в блоке M_S, и размещенная там ранее битовая строка сравнивается со значением B_x элемента x. Таким образом, никогда не требуется проводить какой-либо поиск в таблице.

Пока функция хэширования вычисляется относительно быстро и генерирует хэш-адреса, расположенные достаточно произвольным и равномерным образом, не требуется точного знания того, как именно работает эта функция. Данную мысль можно проиллюстрировать на следующем примере. Предположим, что мы отвели в памяти место под блок M_s размером в 1024 слова (наиболее удобным вариантом для длины блока в двоичных компьютерах является степень 2). Предположим также, что объекты данных, которые нужно разместить в этом блоке, являются строками символов, представленными битовыми строками длиной в два слова каждая. В выделенном блоке памяти можно разместить до 511 различных элементов такого типа. Пусть начальный адрес этого блока памяти равен α. Подходящим хэш-адресом для такой таблицы будет строка I_x длиной девять бит, так как формула $\alpha + 2 \times I_x$ всегда будет генерировать адрес в пределах этого блока. Мы можем вычислить адрес I_x для данной битовой строки B_x длиной в два слова с помощью следующего алгоритма (предполагая, что B_x хранится в словах a и b).

1. Умножаем a на b, получаем c (строка, состоящая также из двух слов).
2. Складываем два слова, представляющих строку c, получаем значение d, представимое одним словом.
3. Возводим d в квадрат, получаем e.
4. Извлекаем центральные девять битов из e, полученная последовательность представляет собой адрес I_x.

Даже самая лучшая функция хэширования не может гарантировать, что для различных элементов данных будут сгенерированы различные хэш-адреса. Хотя желательно, чтобы функция хэширования распределяла адреса по блоку максимально равномерно, тем не менее почти неизбежно происходят так называемые *коллизии*, при которых два элемента получают один и тот же хэш-адрес. Коллизия обычно случается при добавлении некоторого элемента к множеству. Мы обращаемся к блоку памяти, указанному вычисленным для элемента хэш-адресом, и обнаруживаем, что он содержит данные, отличные от тех, которые мы хотим разместить (но для которых алгоритм хэширования сгенерировал точно такой же хэш-адрес). (Иначе говоря, различные элементы x и y получили один и тот же хэш-адрес I_x.) Существует много различных способов разрешения подобных коллизий.

1. *Повторное хэширование.* Мы можем модифицировать исходную строку битов B_x (например, умножив ее на постоянную величину) и затем заново применить к ней алгоритм хэширования, получив тем самым новый хэш-адрес. Если при этом снова случится коллизия, то проводится повторное хэширование до тех пор, пока не отыщется свободное место в блоке памяти или не обнаружится, что точно такой же элемент уже содержится в множестве.
2. *Последовательный поиск.* Начиная от той позиции в блоке, где случилась коллизия, можно начать последовательный (циклический) поиск до тех пор, пока не отыщется свободное место в блоке памяти или не обнаружится, что точно такой же элемент уже содержится в множестве.
3. *Группирование данных.* Вместо непосредственного хранения данных в блоке памяти можно подставить указатели, связанные со списками групп элементов с одинаковыми хэш-адресами. После хэширования значения B_x и по-

лучения указателя на соответствующий список групп осуществляется поиск значения B_x в этом списке, и в случае его отсутствия оно добавляется в конец списка.

Выбор той или иной функции хэширования зависит от свойств представления сохраняемых данных в виде битовых строк. Если функция хэширования достаточно хороша, а таблица заполнена не более чем наполовину, коллизии достаточно редки. Необходимость разрешения коллизий является причиной того, что в приведенном примере таблица для размещения 512 элементов может использоваться только для размещения 511 элементов. В конце концов, алгоритм разрешения коллизий должен найти свободное место, иначе этот процесс никогда не завершится.

6.1.9. Выполняемые объекты данных

В большинстве языков программирования, особенно в компилируемых языках, таких как C или Ada, исходные выполняемые программы и объекты данных, которыми они манипулируют, являются отдельными структурами. Но это не всегда так. В языках, подобных LISP и Prolog, выполняемые операторы могут являться данными, которые доступны программе и которыми она может манипулировать. Например, в языке LISP все данные хранятся в виде списков. Выражение, допустимое в версии Scheme,

```
(define myfunction (cons (a b c) (d e f))
```

просто определяет функцию с именем myfunction как операцию cons, описанную выше. Эта функция хранится как связанный список, во многом похожий на любой другой список.

То же самое можно сказать о языке Prolog и его операции consult, определенной для его списков правил. Хотя управление такими данными достаточно просто осуществляется при использовании описанных здесь списковых структур, возможность запустить программу, модифицировать ее и снова запустить в течение одного ее запуска оказывает большое влияние на структуру ее выполнения. Эта тема будет обсуждаться более подробно в разделе 6.3.

6.2. Абстрактные типы данных

В ранних языках программирования, таких как FORTRAN и COBOL, возможность создавать новые (определяемые программистом) типы данных ограничивалась определением подпрограмм. По мере развития концепции типов данных в новых языках программирования стали появляться средства для определения и реализации полностью абстрактных типов данных. В качестве примера можно привести конструкцию package языка Ada и class языков Java и C++. В этой главе мы изучим более близкий к классическому подход к определяемым программистом типам данных посредством исследования конструкции подпрограмм: как определяются и реализуются процедуры, подпрограммы и функции. В главе 7 мы рассмотрим, как язык Ada расширяет эту концепцию, вводя возможность инкапсуляции дан-

ных, называемых *пакетами* (packages). Мы также разовьем эти идеи, используя понятие *классов* и возможность связывать исполняемые процедуры (например, *методы*) с этими объектами данных для создания того, что сегодня называется *объектно-ориентированными программами.*

6.2.1. Эволюция понятия типов данных

Поскольку обычно аппаратура компьютера не предоставляет возможностей для определения и отслеживания ограничений на типы данных (например, с помощью аппаратных средств невозможно определить различие между строками битов, представляющими вещественное число, символ или целое число), в языках высокого уровня предусматривается набор базовых типов данных, таких как *вещественные* и *целые* числа и *строки символов.* Для того чтобы арифметические операции сложения или умножения не применялись к объектам несоответствующего типа, осуществляется контроль типов. Первоначально *тип данных* определялся как *множество значений*, которые может принимать некоторая переменная. Типы данных непосредственно ассоциировались с отдельными переменными, так что каждое объявление определяло имя переменной и ее тип. Если в программе использовалось несколько массивов, каждый из которых содержал 20 вещественных чисел, каждый массив объявлялся отдельно и для каждого из них повторялось полное описание массива.

Приблизительно в 1970 г. язык Pascal расширил понятие типов данных до общего определения типа, применимого к множеству переменных. *Определение типа* задает структуру объекта данных вместе с возможными связываемыми с ней значениями. Для объявления конкретного объекта данных как принадлежащего определенному типу требуется указать только имя переменной и название этого типа.

В 70-х гг. понятие типов данных расширилось еще дальше и переросло представления о типе просто как о *множестве объектов данных.* В понятие типа данных был включен также *набор операций* над этими объектами. Для элементарных типов данных, таких как целочисленные и вещественные, в языке предусмотрены средства объявления переменных этих типов и набор операций над ними, представляющие *единственный способ* обработки программистом вещественных и целых чисел. Таким образом, представление целых и вещественных чисел в языке эффективно *инкапсулировано* (то есть скрыто от программиста). Программист может использовать целые и вещественные числа, не вникая в то, как именно они реализованы и представлены в памяти. Программист видит лишь название типа и список определенных для этого типа операций, посредством которых можно манипулировать объектами данных этого типа.

Абстракция данных. Для того чтобы распространить введенное понятие инкапсуляции на определяемые программистом типы данных, мы определяем *абстрактные типы данных* как:

1) множество *объектов данных* (обычно используется одно или более определений типов);
2) набор *абстрактных операций* над этими объектами данных;

3) *инкапсуляцию* этих объектов и операций таким образом, чтобы пользователь нового типа данных не мог манипулировать объектами данных этого типа иначе, как только с помощью определенных абстрактных операций. Само определение в целом должно быть инкапсулировано таким образом, чтобы пользователю необходимо было знать лишь название типа и семантику доступных операций. В этой главе мы рассмотрим, как можно разрабатывать абстракции, используя возможности подпрограмм, предоставляемые большинством языков программирования. В главе 7 мы распространим это понятие на такие возможности языка, как, например, реализованные в Ada пакеты (package) или классы (class) в C++ и Smalltalk, которые помогают в реализации процесса инкапсуляции.

6.2.2. Сокрытие информации

Для того чтобы понимать, как разрабатываются языковые средства, с помощью которых можно определять новые типы данных и новые операции, необходимо разобраться с понятием *абстракции*. Если попытаться представить себе все детали большой, а порой и довольно скромных размеров программы, то станет очевидным, что одновременное их рассмотрение выходит за пределы интеллектуальных возможностей человека. Чтобы создать большую программу, приходится использовать принцип «разделяй и властвуй». Программа делится на некоторое множество компонентов, называемых также *модулями*. Каждый модуль выполняет ограниченное количество операций над ограниченным количеством данных.

При проектировании модулей программы обычно придерживаются одного из следующих подходов к декомпозиции программы:

1) по *функциональному* принципу;
2) по признаку используемых *данных*.

Первый из этих двух подходов использовался в 60-е гг. как основная модель разработки программы, и типичная структура подпрограмм, процедуры и функции явились результатом его применения.

Проектирование программ по функциональному принципу считалось оптимальной методикой в течение многих лет. Чтобы понять, в чем недостаток этого подхода, рассмотрим упоминавшийся ранее тип данных *группа*. Чтобы осуществить декомпозицию программы на функциональные единицы, один программист может разработать функции *регистрации*, которые будут создавать *группы*, добавлять *студентов* в *группы* и назначать *преподавателя* для *группы*. Другой программист может взять за основу функции *поддержки класса*, такие как удалить/добавить *студента* из/в *группу*, проставить отметки каждому студенту и разослать счета на оплату каждого курса. В обоих случаях программист должен знать детали того, что собой представляют и *группа*, и *студент*. Однако, создавая модули, которые называются простыми абстракциями, можно избежать большей части отмеченных выше недостатков.

Построение абстракции *группа* и последующее ее использование в других программных модулях — как раз и есть такая простая абстракция. Все, что требуется понимать, чтобы пользоваться абстрактным типом данных, — это его специфика-

ция; детали реализации скрыты и ими можно не интересоваться. Так, программист, разрабатывающий общую программу для распределения студентов по группам, может забыть, как именно организованы группы, и помнить только о том, что можно включать студентов в группы, группы имеют своих преподавателей, пользуются аудиториями с определенными номерами и т. д.

Абстракция настолько распространена в программировании, что часто на нее не обращают внимания. Представление программного обеспечения и аппаратной части в виде слоев, описанное в главе 2, — другой пример абстракции. Блок-схема — это абстракция программной структуры управления на уровне операторов. Методы разработки программ — *пошаговая детализация, структурное, модульное* и *нисходящее программирование* — также связаны с созданием абстракций.

В языках программирования абстракция поддерживается двумя способами. Во-первых, с помощью виртуального компьютера, более мощного и удобного в использовании, чем лежащий в его основе реальный аппаратный компьютер, язык непосредственно предоставляет набор полезных абстракций, которые мы воспринимаем как свойства этого языка. Во-вторых, язык программирования предоставляет средства, позволяющие самому программисту конструировать абстракции, которые совместно формируют виртуальный компьютер, определяемый конкретной программой. Такими средствами в различных языках являются подпрограммы, библиотеки подпрограмм, определения типов, классы и пакеты.

Термин *сокрытие информации* используется для обозначения основного принципа, лежащего в основе проектирования абстракций, определяемых программистом: каждый такой программный компонент должен насколько возможно скрывать от пользователей информацию о своем внутреннем устройстве. Например, встроенная в язык функция вычисления квадратного корня является удачной абстрактной операцией, поскольку она скрывает от пользователя детали представления чисел и алгоритм вычисления квадратного корня. Аналогично определяемый пользователем тип данных может служить примером удачной абстракции, если в итоге его можно использовать, не зная конкретного представления объектов этого типа и алгоритмов, которые применяются для выполнения операций над объектами этого типа.

Когда информация *инкапсулирована* в абстракцию, это означает, что пользователь:

1) не нуждается в скрытой информации для того, чтобы пользоваться абстракцией;
2) не имеет возможности (ему не позволено) непосредственно использовать скрытую информацию или как-либо манипулировать ею, даже если бы он и захотел этого.

Например, целочисленный тип данных в таких языках программирования, как FORTRAN или C не только скрывает детали представления целых чисел в памяти, но инкапсулирует это представление таким образом, что программист не имеет возможности манипулировать отдельными битами в представлении целого числа (за исключением тех случаев, когда в механизме инкапсуляции данного языка допущены какие-либо дефекты, делающие возможным несанкционированный доступ к битовому представлению объектов данных). Но пользователю языков FORTRAN

или C гораздо сложнее инкапсулировать представление нового типа данных. Хотя можно создать набор подпрограмм, которые позволяют создать тип *группа* и манипулировать им как абстракцией (то есть обеспечить сокрытие информации), при этом невозможно инкапсулировать представление данных, использованное при его реализации, таким образом, чтобы пользователь абстракции не смог написать другую подпрограмму, которая будет манипулировать *группой* непредусмотренным образом. Например, если список записавшихся на курс студентов будет представлен в виде линейного массива целых чисел, то можно написать подпрограмму, которая к каждому из этих чисел прибавит 3. В терминах данной абстракции, где целые числа представляют идентификационные номера студентов, эта операция не имеет смысла, но в терминах выбранного способа представления (линейный массив целых чисел) эта операция абсолютно правомерна.

Инкапсуляция имеет большое значение при модификации программ, поскольку она существенно упрощает этот процесс. Если бы нам удалось инкапсулировать тип данных *группа*, то можно было бы изменять представление объектов этого типа в любой момент, просто модифицируя подпрограммы, которые манипулируют объектами этого типа, так чтобы они могли работать с их новым представлением взамен старого. Но если тип данных *группа* не инкапсулирован, то другие части программы могут продолжать использовать его старое представление. Поэтому любые изменения в представлении этого типа данных привели бы к тому, что эти другие части программы не смогли функционировать. Часто очень трудно определить все подпрограммы, на которые (в отсутствие инкапсуляции) может повлиять изменение способа представления конкретных объектов данных. Поэтому изменения в их представлении могут повлечь за собой трудноуловимые ошибки в тех частях программы, на которые, казалось бы, это изменение не может повлиять.

Подпрограммы формируют основной механизм инкапсуляции, который присутствует почти во всех языках. Современные механизмы, которые позволяют осуществлять инкапсуляцию определения целых типов данных, из тех языков, которые описаны в нашей книге, присутствуют только в Ada и C++[1]. Заметим, что ***сокрытие информации*** — это прежде всего вопрос ***проектирования программ***; сокрытие информации возможно в любой правильно спроектированной программе вне зависимости от используемого языка программирования. ***Инкапсуляция***, однако, — это прежде всего вопрос разработки конкретного языка; абстракция может быть эффективно инкапсулирована только в том случае, если язык запрещает доступ к скрытой в абстракции информации.

6.3. Инкапсуляция при помощи подпрограмм

Подпрограмма — это абстрактная операция, определяемая программистом. Подпрограммы являются теми базовыми блоками, из которых строится большинство программ, и почти в каждом языке можно найти средства для их определения и вызова.

---

[1] Авторы забыли упомянуть язык Java. — *Примеч. науч. ред.*

Для нас здесь важны два взгляда на подпрограммы. На уровне *проектирования программы* подпрограмма нас интересует в том смысле, в котором она представляет абстрактную операцию, определенную программистом, — в противоположность элементарным операциям, встроенным в язык. На уровне *разработки языка* нас интересует разработка и реализация общих средств определения и вызова подпрограммы. Хотя эти два взгляда и пересекаются, удобнее рассматривать их по отдельности.

6.3.1. Подпрограммы как абстрактные операции

Как и в случае элементарных операций, определение подпрограммы состоит из двух частей: *спецификации* и *реализации*. Но в случае подпрограммы обе эти части определяются программистом при ее описании.

Спецификация подпрограммы. Поскольку подпрограмма является абстрактной операцией, ее спецификация должна быть понятна вне зависимости от реализации. Спецификация подпрограммы, по сути, ничем не отличается от спецификации элементарной операции и включает:

1) *имя* подпрограммы;
2) *сигнатуру* (называемую также прототипом) подпрограммы, которая задает количество *аргументов*, порядок их следования, тип данных для каждого аргумента, а также количество *результатов*, их порядок и тип данных для каждого результата;
3) *действие*, выполняемое подпрограммой (иначе говоря, описание той функции, которую она вычисляет).

Подпрограмма представляет собой некоторую математическую функцию, которая отображает каждый конкретный набор аргументов в некоторый набор результатов. Если подпрограмма явно возвращает только один результирующий объект данных, она обычно называется *подпрограммой-функцией* (или просто *функцией*). Типичным примером синтаксиса для спецификации функции является следующее выражение на C:

```
float FN(float X, int Y)
```

которое определяет сигнатуру функции как

```
FN : вещественное × целое → вещественное
```

Заметим, что в спецификацию также входят имена X и Y, которые можно использовать для обращения к аргументам внутри подпрограммы. Эти аргументы называются *формальными параметрами*. Общие вопросы передачи параметров в подпрограмму обсуждаются более подробно в разделе 9.3. Кроме того, в некоторых языках имеется специальное ключевое слово, например procedure или function, которое используется в объявлении подпрограммы, как, например, в языке Pascal:

```
function FN(X: real; Y: integer): real;
```

Если подпрограмма возвращает более одного значения или если ее действия сводятся к модификации своих аргументов вместо явного возвращения результата, она обычно называется *процедурой* или *подпрограммой*. Типичным примером синтаксиса для спецификации подпрограммы является следующее выражение языка C:

```
void Sub(float X, int Y, float *Z, int *W);
```

В приведенной спецификации ключевое слово void указывает на пустую функцию, подпрограмму, которая не возвращает никакого результата. Символ * перед именем формального параметра означает, что этот параметр может являться результирующим значением или что это аргумент, значение которого может быть модифицировано в ходе выполнения подпрограммы. (Фактически такие аргументы, как будет показано в разделе 9.3 являются переменными-указателями.) Эта же спецификация в языке Ada более наглядно выражает указанные различия в аргументах:

```
procedure Sub(X: in REAL; Y: in integer;
          Z: in out REAL; W: out BOOLEAN)
```

Этот заголовок определяет подпрограмму со следующей сигнатурой:

Sub : $вещественное_1 \times целое \times вещественное_2 \rightarrow вещественное_3 \times$ булево значение

Ключевые слова in, out и in out указывают на три способа вызова аргументов в подпрограмме: in обозначает, что аргумент является входным и не модифицируется подпрограммой, in out обозначает, что аргумент является входным, но может быть модифицирован при выполнении подпрограммы, и out обозначает результат. Более подробно эта тема обсуждается в разделе 9.3.

Хотя подпрограмма и представляет некоторую математическую функцию, при попытке определить, какая в точности вычисляется функция, возникают некоторые проблемы.

1. В подпрограмме могут использоваться *неявные аргументы* в виде ссылок на нелокальные переменные.
2. При выполнении подпрограммы могут возникать *неявные результаты* (*побочные эффекты*), возвращаемые в виде изменений значений нелокальных переменных или в виде изменений значений ее аргументов типа in out.
3. Для некоторых возможных значений аргументов подпрограмма может быть не определена, так что она не завершит свои вычисления как обычно, если ей будут переданы подобные аргументы, а вместо этого передаст управление какому-либо внешнему обработчику исключений или вдруг неожиданно вообще завершит выполнение всей программы.
4. Подпрограмма может быть *чувствительна к предыстории ее вызовов*, так что результаты ее выполнения зависят от последовательности переданных в нее аргументов на протяжении всей истории ее вызовов, а не только от аргументов, переданных в конкретном вызове. Чувствительность к предыстории может быть вызвана сохранением значений каких-либо локальных данных этой подпрограммы в промежутках между обращениями к ней.

Реализация подпрограмм. Подпрограмма представляет операцию слоя виртуального компьютера, который сконструирован программистом, и, следовательно, подпрограмма реализуется при помощи структур данных и операций, предоставляемых языком программирования. Реализация определяется *телом* подпрограммы, которое состоит из *объявлений локальных данных*, определяющих структуры данных, используемых подпрограммой, и *операторов*, задающих действия, которые должна выполнить подпрограмма. Объявления и операторы обычно инкапсулированы, так что ни локальные данные, ни операторы по отдельности не доступны пользователю подпрограммы; пользователь может только вызвать подпрограмму с конк-

ретным набором аргументов и получить вычисленные результаты. Синтаксис языка C для тела подпрограммы является достаточно типичным:

```
float FN(float X, int Y)           - сигнатура подпрограммы
   {float M(10); int N;            - объявления локальных объектов данных
   .                               - последовательность операторов,
   .}                              определяющая действия подпрограммы
```

В некоторых языках (например, в Pascal и Ada, но не в C) в тело подпрограммы могут входить определения других подпрограмм, которые представляют собой определяемые программистом операции, используемые только в пределах содержащей их более крупной подпрограммы. Эти локальные подпрограммы также инкапсулированы, так что к ним можно обращаться только изнутри той подпрограммы, в которой они определены, но не извне.

Каждый вызов подпрограммы требует передачи ей аргументов правильного типа в соответствии с определенной ее спецификацией. Также должны быть известны типы результатов, возвращаемых подпрограммой. Контроль типов в случае подпрограмм аналогичен контролю типов, который применяется для элементарных операций. Контроль типов может осуществляться статически во время трансляции, если для типов аргументов и результатов всех подпрограмм заданы объявления. Альтернативой является динамический контроль типов, происходящий во время выполнения программы. *Приведение* аргумента к правильному типу также может осуществляться автоматически, если подобная операция предусмотрена реализацией языка. Методы реализации и связанные с этим вопросы являются прямым продолжением и обобщением концепций, представленных в главе 5 для элементарных операций. Основное различие заключается в том, что в случае подпрограмм программист должен явным образом объявлять информацию о типах аргументов и результатов, в то время как для элементарных операций эта информация передается неявным образом. Но когда эта информация предоставлена, то дальнейшие аспекты контроля типов трактуются аналогично.

6.3.2. Определение и вызов подпрограмм

Разработка средств для определения и вызова подпрограмм является одной из центральных задач — возможно, самой главной задачей — при разработке большинства языков. Большая часть структуры реализации определяется именно структурой подпрограмм. Здесь мы обсудим некоторые общие концепции реализации подпрограмм. В главе 7 мы рассмотрим методы для обеспечения полной инкапсуляции и сокрытия информации о данных.

Определение и активизация подпрограмм

Определение подпрограммы является статическим свойством программы. Если во время выполнения программы вызывается (или *запускается*) подпрограмма, то создается *активация* подпрограммы. По завершении выполнения подпрограммы активация разрушается. Если подпрограмма вызывается следующий раз, создается новая активация. Во время выполнения программы по единственному определению подпрограммы может быть создано много активаций. Определение подпрограммы служит *шаблоном* для создания активаций во время выполнения программы.

Важно понимать различие между определением и активацией подпрограммы. Определение подпрограммы — это то, что присутствует в тексте программы и является единственной доступной информацией во время трансляции (например, во время трансляции известен тип переменных подпрограммы, но нет никакой информации об их значениях или местоположении в памяти (*r*-значение или *l*-значение)). Активации подпрограмм существуют только во время выполнения программы. При этом может быть выполнен код для доступа к *r*-значению или *l*-значению переменной, но тип переменной может быть недоступен, если только транслятор не сохранил эту информацию в дескрипторе переменной.

Это различие очень похоже на различие между определением типа и объектом данных этого типа, которое рассматривается более подробно в следующем разделе. Определение типа используется как шаблон для определения размера и структуры объектов данных этого типа. Однако сами объекты данных обычно создаются во время выполнения программы либо на входе в подпрограмму, либо при выполнении операции создания объектов данных, такой, например, как `malloc`. Использование операции `malloc` для создания новых объектов данных во время выполнения подпрограммы соответствует использованию вызова (call) подпрограммы для создания новых активаций подпрограмм. Фактически активация подпрограммы является некоторым типом объекта данных. Ей соответствует определенный блок памяти, в котором содержатся некоторые компоненты данных, относящиеся к активации подпрограммы. При создании активации ей должна быть выделена область памяти, которая освобождается при ее разрушении. Таким образом, активация имеет определенное время жизни — промежуток времени между создающим ее вызовом подпрограммы и выходом из подпрограммы, когда активация разрушается. Однако существуют понятия, касающиеся активации подпрограмм, которые не имеют прямого аналога среди других объектов данных (например, понятие *выполнения* активации и понятие ссылки и модификации других объектов данных во время этого выполнения). По причине этих различий, а также из-за того, что интуитивно чувствуется большая разница между подпрограммами и прочими объектами данных в большинстве языков программирования, мы не используем в этой книге термин *объект данных* применительно к активации подпрограммы.

Реализация определения и вызова подпрограммы

Рассмотрим определение подпрограммы в языке С (листинг 6.1). В нем определены все компоненты, необходимые для активации подпрограммы во время выполнения.

1. Первая строка — строка сигнатуры функции `FN` — предоставляет информацию о необходимой памяти для ее *параметров* (объекты данных `X` и `Y`) и для ее *вычисляемого значения* (объект данных типа `float`).
2. Объявления определяют способ представления в памяти *локальных переменных* (массив `M` и переменная `N`).
3. Определяется представление в памяти *литералов* и *именованных констант*: `initval` — это именованная константа со значением `2`, `finalval` —именованная константа со значением `10`, а `10` и `20` являются литералами.
4. Определяется объем памяти для *исполняемого кода*, который генерируется при трансляции операторов, содержащихся в теле подпрограммы.

Листинг 6.1. Определение подпрограммы в языке С

```
float FN(float X, int Y)
      {const initval = 2;
      #define finalval 10
      float M(10); int N;
      N = initval;
      if(N<finalval){…}
      return(20 * X + M(N));}
```

Отметим одну важную особенность языка С. Атрибут const информирует компилятор языка С о том, что объект данных initval — это числовая константа со значением 2. Тогда как конструкция #define finalval 10 является макрокомандой препроцессора, которая преобразует каждый встретившийся в тексте программы идентификатор finalval в символ 10. Таким образом, компилятор языка С даже не сталкивается с идентификатором finalval, так как везде он будет заменен на числовой литерал 10. Практический эффект для выполнения подпрограммы одинаков в обоих случаях, но смысл конструкций const и #define совершенно различен. У объекта данных initval имеется *l*-значение, чье *r*-значение равно 2. У объекта finalval имеется только *r*-значение, равное 10.

Определение подпрограммы позволяет еще во время трансляции организовать представление в памяти всех используемых объектов данных и определить исполняемый код. Результатом трансляции будет шаблон, который используется для создания активации каждый раз при вызове подпрограммы. На рис. 6.11 показано, как определение подпрограммы преобразуется в шаблон, используемый во время выполнения подпрограммы.

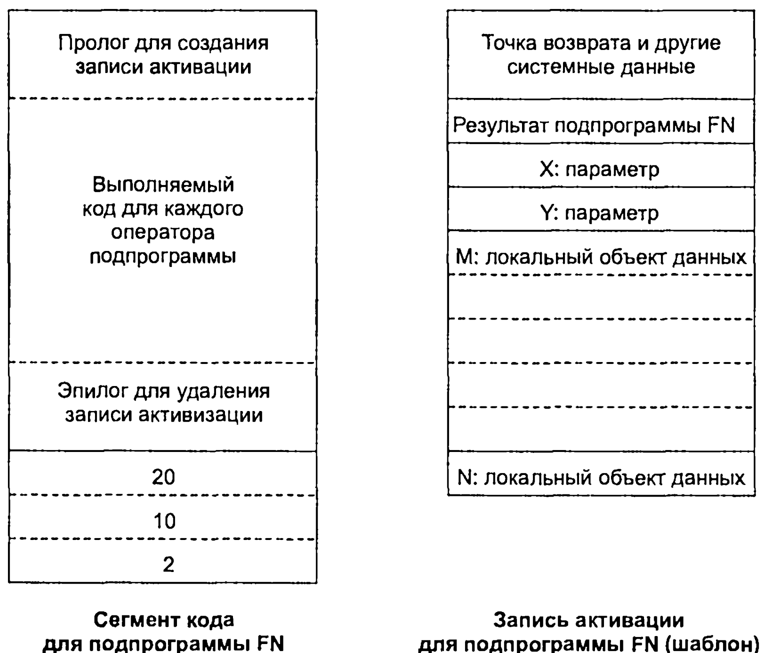


Рис. 6.11. Структура активации подпрограммы

Для конструирования конкретной активации подпрограммы на основе ее шаблона следует скопировать весь этот шаблон в новую область памяти. Однако вместо того, чтобы делать полную копию, гораздо лучше разбить шаблон на две части:

- статическую часть, которая называется *сегментом кода* и состоит из констант и выполняемого кода. Эта часть не изменяется в процессе выполнения подпрограммы и, следовательно, может быть использована повторно для других активаций;
- динамическую часть, которая называется *записью активации* и состоит из параметров, результатов функций и локальных данных, а также других вспомогательных данных, зависящих от реализации языка (например, временных областей памяти, точек возврата из подпрограммы и связей для ссылок на нелокальные переменные — более подробно эта тема обсуждается в разделе 9.2). Структура этой части также одинакова для всех активаций данной подпрограммы, но конкретные значения данных различаются. Следовательно, для каждой активации обязательно требуется своя запись активации.

Итоговая структура во время выполнения подпрограммы показана на рис. 6.12. Во время выполнения программы для каждой подпрограммы в памяти хранится один сегмент кода. Записи активации динамически создаются при обращении к подпрограмме и уничтожаются каждый раз при завершении ее выполнения.

Размер и структура записи активации, как правило, могут быть определены во время трансляции (например, компилятор или транслятор может определить количество компонентов, необходимое для хранения необходимой информации внутри записи активации, а также местоположение каждого компонента). Доступ к компонентам можно осуществить, используя базовый адрес и вычисляемый относительно него сдвиг, как это описано в разделе 6.1.3 для обычных записей. По этой причине запись активации представляется в памяти точно так же, как и любая другая запись. Для создания новой записи активации во время выполнения программы требуется знать размер блока памяти, но не его внутреннюю структуру (так как сдвиги внутри этого блока уже вычислены при трансляции). Поэтому для вычисления формулы доступа во время выполнения программы нужно знать только базовый адрес блока памяти. Вместо того чтобы хранить весь шаблон записи активации во время выполнения программы, нужно знать только размер этой записи, который будет использоваться операцией вызова подпрограммы. Управление памятью при вызове подпрограммы и выходе из нее сводится только к выделению под запись активации блока памяти соответствующего размера в момент обращения к подпрограмме и его освобождения по завершении выполнения подпрограммы. Как будет показано в главе 9, обычно для размещения и удаления из памяти записей активации используется простой стек.

Когда вызывается подпрограмма, осуществляется ряд скрытых действий, связанных с созданием записи активации, передачей параметров, формированием ссылок на нелокальные переменные и другими подобными вспомогательными задачами. Эти действия должны быть выполнены до того, как начнет выполняться код, представляющий операторы тела подпрограммы. За выполнение этого *пролога* обычно отвечает транслятор, который вставляет соответствующий код, выполняющий все необходимые действия, перед началом кода самой подпрограммы. По

завершении выполнения подпрограммы требуется осуществить аналогичный ряд вспомогательных действий, связанных с возвращением результатов и освобождением выделенной области памяти под запись активации. Этот *эпилог* состоит из набора инструкций, вставляемого транслятором в конец выполняемого кода подпрограммы. Подробности мы рассмотрим позже в этой главе.

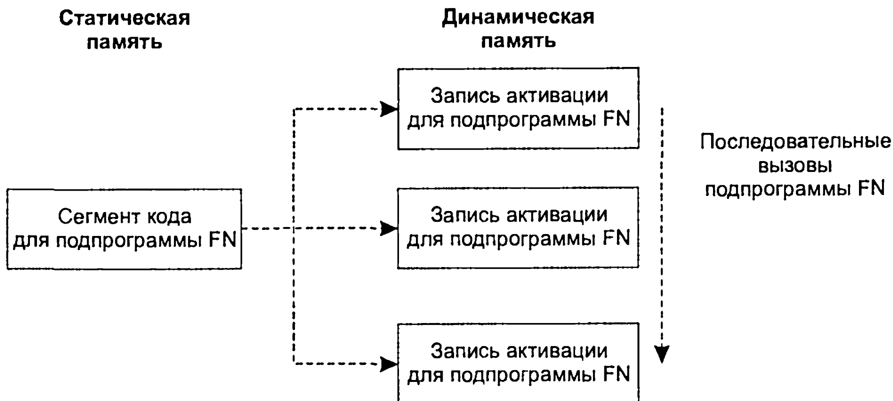


Рис. 6.12. Общий сегмент кода и различные записи активации для подпрограммы

Общие подпрограммы

В спецификации подпрограммы обычно указываются количество, порядок следования и тип данных ее формальных параметров. *Общая подпрограмма* — это подпрограмма, имеющая одно имя, но несколько различных определений, характеризующихся различными сигнатурами. Говорят, что имя общей подпрограммы *перегружено*. Общая концепция общих операций и перегруженных имен обсуждалась в главе 5 на примере элементарных операций. Подпрограммы с этой точки зрения также могут быть общими. Например, при создании большой программы распределения студентов по различным курсам лекций в университете в числе других подпрограмм могут потребоваться следующие: одна, которая включает какую-либо группу в таблицу со списком групп, и другая, которая вносит фамилию студента в список группы. Обе эти подпрограммы могут быть названы Enter:

```
procedure Enter(Student: in integer;
                Sect: in out Section) is
   begin - здесь помещаются операторы включения студентов в список группы
   end;
procedure Enter(S: in Section;
                Tab: in out Classlist) is
   begin - здесь помещаются операторы включения групп в список
   end;
```

Имя Enter является перегруженным и становится именем общей подпрограммы Enter. Когда происходит вызов процедуры Enter (Enter(A, B)), транслятор сопоставляет типы фактических параметров A и B с типами формальных параметров в объявлениях обеих процедур Enter. Если переменная A окажется типа integer, это будет означать, что следует вызывать первую из приведенных в листинге процедур. Если же тип переменной A является section, то — вторую процедуру. По-

скольку это разрешение имен осуществляется еще во время трансляции, то в таких языках, как Ada, перегрузка имени не оказывает влияния на организацию этапа выполнения программ, написанных на этом языке. Если вопрос с перегрузкой решен, то далее трансляция вызова одной из родственных подпрограмм происходит так же, как и для любой другой подпрограммы.

В FORTRAN 90 перегруженные процедуры определяются в специальном блоке INTERFACE, который называется интерфейсным блоком. Наш пример с двумя процедурами включения в список студента или группы будет выглядеть следующим образом:

Листинг 6.2. Интерфейсный блок в FORTRAN 90

```
INTERFACE ENTER
      SUBROUTINE ENTER_STUDENT(STUDENT, SECT)
         INTEGER:: STUDENT
         SECTION:: SECT
         END SUBROUTINE ENTER_STUDENT
      SUBROUTINE ENTER_SECTION(S, TAB)
         SECTION:: S
         CLASSLIST:: TAB
         END SUBROUTINE ENTER_SECTION
END INTERFACE ENTER
```

В этом примере подпрограмма ENTER определена таким образом, что может получать в качестве фактического параметра либо объект типа integer, либо объект типа section. Следующим шагом в развитии этой концепции является включение непосредственно самого типа в параметры подпрограммы, как это сделано в языке ML с помощью механизма полиморфных типов.

Возможность определять общие подпрограммы не требует каких-либо значительных изменений языка или его возможной реализации, но оказывает большое влияние на использование языка. Поэтому перегрузка имени подпрограммы, которая в языке Ada является просто дополнительной возможностью, в ML уже становится одной из существенных особенностей программирования на этом языке. Более подробно это обсуждается в главе 7.

6.3.3. Определения подпрограмм как объектов данных

В большинстве компилируемых языков, таких как С, С++ и Java, определение подпрограммы отделено от ее выполнения. Сначала исходная программа обрабатывается компилятором и преобразуется в исполняемый код. Во время выполнения статическая часть определения подпрограммы недоступна и невидима. Но в таких языках, как LISP, Perl и Prolog (которые обычно реализуются при помощи программных интерпретаторов), как правило, между этими двумя фазами различие отсутствует. Возможности этих языков позволяют обращаться с *определениями* подпрограмм, как с некими объектами данных времени выполнения.

Трансляция — это операция, которая получает определение подпрограммы в виде строки символов и создает объект данных времени выполнения, представляющий это определение. *Выполнение* — это операция, которая получает определение подпрограммы в форме объекта данных времени выполнения, создает на его основе

активацию и выполняет ее. Операция выполнения запускается при помощи элементарной команды вызова подпрограмм, но операция трансляции часто рассматривается как отдельная *метаоперация*, которая выполняется для всех подпрограмм до начала выполнения основной программы. В языках LISP и Prolog трансляция, однако, является операцией, которую можно инициализировать во время выполнения основной программы и которая в качестве аргумента получает символьную строку (определение подпрограммы), а в качестве результата выдает исполняемый код тела подпрограммы. В этих языках имеется специальная операция `define`, аргументами которой являются тело и спецификация подпрограммы, а результатом — готовое к использованию определение подпрограммы (например, операции `define` в LISP, `consult` в Prolog).

Таким образом, в обоих этих языках можно запускать программу, не имея в наличии конкретной подпрограммы. Во время выполнения программы тело подпрограммы может быть считано из какого-либо внешнего источника или создано как объект данных символьного типа и затем оттранслировано в исполняемую форму. Затем применяется операция `define` для определения имени подпрограммы и параметров, используемых в теле подпрограммы. В результате получается полное определение подпрограммы. Впоследствии эта подпрограмма может вызываться по мере надобности. Определение подпрограммы позже может быть изменено. Таким образом, в таких языках определения подпрограмм становятся полноценными объектами данных.

6.4. Определения типов

При определении совершенно нового абстрактного типа данных необходим некоторый механизм для определения класса объектов данных. В таких языках, как С, Pascal и Ada, этот механизм называется *определением типа* (заметим, что это определение типа все же не определяет полностью абстрактный тип данных, поскольку в него не входит определение *операций*, применимых к вновь созданному типу).

В определении типа задается *имя* этого типа и объявления, которые описывают структуру класса объектов данных. Имя типа становится затем именем этого класса объектов данных. Когда нам в программе потребуется использовать объект данных этого типа, нужно будет только указать имя типа, а повторять полное описание структуры этого объекта не понадобится. Если в некоторой программе на языке Pascal используются три записи `A`, `B` и `C`, имеющие одну структуру (например, эти записи могут представлять рациональные числа), то программа может содержать следующее определение типа:

```
type Rational = record
            numerator: integer;
            denominator: integer
            end
```

которое используется в объявлении

```
var A, B, C: Rational;
```

Таким образом, определение структуры объекта данных типа `Rational` задается один раз, а не повторяется для каждой из трех записей `A`, `B` и `C`.

В языке С похожая ситуация, но несколько более сложная по сравнению с рассмотренным примером из Pascal. Новые типы данных могут определяться только при помощи конструкции srtuct. Следовательно, тип Rational может быть задан в С следующим образом:

```
struct RationalType
            {int numerator:
            int denominator: }
```

Использование нового типа в языке С тем не менее требует указания в объявлении конструкции srtuct. В программе это будет выглядеть следующим образом:

```
struct RationalType A. B. C:
```

На самом деле то, что получилось, не вполне отвечает принципам абстракции данных. Дело в том, что нам хотелось бы внедрить новый тип данных в язык таким образом, чтобы он синтаксически и семантически ничем не отличался от элементарных типов данных, встроенных в язык. Следовательно, наличие ключевого слова struct нежелательно. Причина же в том, что язык С достаточно старый — он начал развиваться еще в начале 70-х гг., когда понятия инкапсуляции и сокрытия информации еще не были полностью сформированы. К счастью, в С предусмотрена конструкция typedef, реализующая выход из этого положения:

```
typedef определение_типа имя
```

Фактически здесь происходит замена соответствующего определения типа (определение_типа) на новое имя типа (имя), почти как в макроподстановке. Таким образом, на языке С определение типа Rational можно записать следующим образом:

```
typedef struct RationalType
            {int numerator:
            int denominator: } Rational:
```

Затем можно использовать это определение в объявлении:

```
Rational A. B. C:
```

Таким образом, мы получаем такой же синтаксис для определения объектов нового типа, как и в Pascal. Rational здесь обозначает новый тип данных, определенный как struct RationalType. Важно помнить, что в языке С новый тип данных генерируется при помощи конструкции struct, в то время как конструкция typedef, несмотря на свое название, играет роль макроподстановки для определения типа. Но чаще все-таки конструкция typedef применяется для создания новых типов, и в большинстве языков она используется как объявление нового типа данных.

Помимо упрощения структуры подпрограммы определение типов имеет и другие преимущества для программиста. Если по какой-либо причине нужно будет модифицировать определение типа Rational, то это можно будет сделать в одном только месте и не придется отыскивать и переделывать определения для каждого экземпляра переменных этого типа. Также в случае, если переменная нового типа передается в качестве параметра какой-либо подпрограмме, обычно в ней должно присутствовать описание этого нового типа. Если у нас имеется определение типа, то мы просто указываем, что данная переменная принадлежит этому типу, вместо того чтобы повторять все описание структуры этих объектов заново.

Определение типа используется как *шаблон* для конструирования объектов данных во время выполнения программы. Объект данных нового типа может быть создан или на входе в подпрограмму (если объявление переменной этого типа присутствует среди объявлений других локальных данных в этой подпрограмме), или динамически во время выполнения подпрограммы при помощи некоторой специальной операции создания (например, malloc, которая обсуждалась в разделе 5.3.2). Во втором случае доступ к созданному объекту осуществляется при помощи указателей. Использование определения типа в качестве шаблона аналогично использованию определения подпрограммы, описанному в предыдущем разделе.

Определение типа позволяет отделить определение *структуры* объекта данных от определения тех точек во время выполнения программы, в которых эти объекты *создаются*. Заметим также, что у нас появляются новые способы *инкапсуляции* и *сокрытия информации*. Переменную в подпрограмме можно определить как относящуюся к некоторому типу, просто указав имя этого типа. Определение типа фактически скрывает внутреннюю структуру объекта данных этого типа. Если подпрограмма просто создает объекты данных некоторого типа, используя его имя, но никогда не осуществляет доступ к внутренним компонентам этого объекта данных, то подпрограмма становится фактически независимой от конкретной структуры, объявленной в определении типа. То есть это определение можно модифицировать, не меняя при этом подпрограмму. Если идеология языка позволяет ввести ограничения на то, что лишь некоторые специальным образом указанные подпрограммы смогут иметь доступ к внутренним компонентам объектов данных, а все остальные подпрограммы эти объекты будут воспринимать как единое целое, то определение типа фактически инкапсулирует структуру объектов данных этого типа. В следующем разделе мы рассмотрим, как подобная инкапсуляция позволяет программисту сконструировать полный абстрактный тип данных.

Реализация. Информация, содержащаяся в объявлении переменной, в первую очередь используется во время трансляции для определения способа представления в памяти соответствующего объекта данных, управления ресурсами памяти и контроля типов. Во время выполнения программы объявления отсутствуют; они нужны только для того, чтобы правильным образом создать во время выполнения необходимые объекты данных. Аналогично и определение типа используется только во время трансляции. Транслятор языка заносит информацию, полученную из определения типа, в специальную таблицу и, когда имя типа встречается в последующих объявлениях, использует данные из этой таблицы для того, чтобы создать соответствующий этому типу исполняемый код для создания объектов данных этого типа и манипулирования ими. Определение типа позволяет осуществлять некоторые аспекты трансляции, такие как определение способов представления объектов в памяти, только один раз, а не повторять многократно при каждом упоминании этого типа в объявлениях. Тем не менее наличие в языке средств для определения типов, как правило, не влияет на способ реализации языка.

6.4.1. Эквивалентность типов

Проверка типов, статическая или динамическая, включает в себя сравнение типа данных фактического аргумента, переданного операции и ожидаемого для этой

операции типа данных. Если эти типы одинаковы, то аргумент принимается и операция выполняется. Если эти типы различны, то либо создавшаяся ситуация расценивается как ошибка, либо применяется приведение типа аргумента к ожидаемому.

Понятие эквивалентности типов поднимает вопрос о двух связанных понятиях.

1. Что означает высказывание «два типа одинаковы»?
2. Что означает высказывание «объекты данных одного и того же типа равны между собой»?

Первое понятие из области проблем типов данных. Если мы можем определить тип данных статически, то мы имеем дело со строго типизированным языком. Второе понятие относится к области семантики и связано с определением *r*-значения объекта данных.

Равенство типов. Когда тип X равен типу Y? Мы по большей части старались игнорировать этот тонкий вопрос при обсуждении элементарных типов данных, таких как целые числа или массивы (хотя этот вопрос затрагивался, когда речь шла о подтипах элементарных типов). Определения типов требуют ясного ответа на этот вопрос, если мы хотим, чтобы язык был четко и ясно определен.

Рассмотрим определения типов и объявления в программе, приведенной в листинге 6.3. Наш вопрос заключается в следующем: одинаковы ли типы данных переменных X, Y, Z и A. В случае положительного ответа на этот вопрос операция присваивания X := Y и вызов подпрограммы Sub() с параметром Y будут иметь вполне законное основание, в противном случае оба эти действия становятся недействительными.

Листинг 6.3. Равенство типов

```
program main(input, output);
type Vect1: array [1..10] of real;
         Vect2: array[1..10]of real;
var X,Z: Vect1; Y: Vect2;
procedure Sub(A: Vect1);
         ...
         end;
begin                                   - главная программа
      X := Y;
      Sub(Y)
end
```

Существует два способа решения этой проблемы: *эквивалентность имен* и *структурная эквивалентность*.

Эквивалентность имен. Два типа данных считаются эквивалентными, если они имеют одинаковое имя. Таким образом, Vect1 и Vect2 считаются различными типами, даже если объекты данных этих типов имеют одинаковую структуру. Тогда правильной будет операция присваивания X := Z, но не операция X := Y. Метод эквивалентности имен используется в языках Ada, C++ и для параметров подпрограмм в Pascal (для остальных случаев в Pascal он не применяется).

Эквивалентность имен как способ определения эквивалентности типов имеет следующие недостатки.

1. Тип каждого объекта, используемого в операции присваивания, должен иметь определенное имя. *Анонимные типы* не допускаются. Декларация языка Pascal

   ```
   var W: array [1..10] of real
   ```

вполне однозначно определяет тип массива W, но переменная W не может быть передана подпрограмме в качестве фактического параметра, поскольку ее тип не имеет имени.

2. Единственное определение типа должно использоваться во всей программе или в большей ее части, поскольку тип объекта данных, переданного в качестве фактического параметра по цепочке подпрограмм, не может заново определяться в каждой подпрограмме — должно использоваться единое глобальное определение типа. Средствами для этого являются определения классов в C++, файлы включения в C и имена пакетов спецификаций в Ada.

Структурная эквивалентность. Два типа данных считаются эквивалентными, если соответствующие им объекты данных имеют одинаковую *внутреннюю структуру*, то есть состоят из одинаковых компонентов. Обычно это означает, что для обоих классов объектов можно использовать один и тот же способ их представления в памяти. Например, Vect1 и Vect2 при таком подходе будут считаться эквивалентными типами, поскольку каждый объект данных типа Vect1 и каждый объект данных типа Vect2 имеют в точности одинаковое число компонентов эквивалентных типов, расположенных в одном и том же порядке. Представление объектов данных каждого типа в памяти одинаково, поэтому для выборки их компонентов можно использовать одну и ту же формулу доступа; вообще, реализации этих типов данных во время выполнения полностью идентичны.

Структурная эквивалентность типов лишена недостатков, характерных для равенства типов на основе эквивалентности имен, но имеет целый ряд своих недостатков.

1. При попытке применить принцип структурной эквивалентности типов возникают некоторые проблемы. Например, в случае записей должны ли совпадать имена компонентов или достаточно только совпадения их типов, количества и порядка следования? Если же имена компонентов записей должны совпадать, требуется ли тогда совпадение их порядка? Должны ли совпадать диапазоны значений индексов для массивов или достаточно только совпадения числа компонентов? Должны ли совпадать литералы и порядок их следования для двух перечислений?

2. Две переменные случайно могут оказаться структурно эквивалентными, хотя программист определял их как принадлежащие к различным типам данных, в чем можно убедиться на следующем простом примере:

```
type Meters = integer;
         Liters = integer;
var Len: Meters;
         Vol: Liters;
```

   Переменные Vect1 и Vect2 имеют структурно эквивалентные типы, и поэтому такая ошибка, как, например, вычисление суммы Vect1 + Vect2 не будет обнаружена при статической проверке типов. Когда несколько программистов пишут одну большую программу, непреднамеренное совпадение типов различных объектов может привести к тому, что будут утрачены все преимущества статической проверки типов, так как многие ошибки определения типов могут остаться незамеченными.

3. Частая проверка эквивалентности типов в случае сложных определений их структур может значительно увеличить стоимость трансляции.

Вопросы, связанные с выбором способа определения эквивалентности типов, играют важную роль в идеологии таких языков, как Ada и Pascal, где эквивалентность типов имеет большое значение. В более старых языках, таких как FORTRAN, COBOL и PL/I, где определения пользовательских типов отсутствуют, используется некое подобие структурной эквивалентности. В языке Pascal вопрос эквивалентности типов ставится, но он не решается последовательно с использованием только одного из рассмотренных способов. В языке С используется структурная эквивалентность типов. В языке C++, как было сказано ранее, используется эквивалентность имен (в этом — одно из тонких различий между языками С и C++). В языке Ada в настоящее время используется эквивалентность имен, но пока этот вопрос еще находится в стадии исследований (см. задачи и ссылки на рекомендуемую литературу в конце этой главы).

Равенство объектов данных. Когда компилятор определил, что два объекта данных имеют один и тот же тип, можно поставить вопрос о равенстве этих объектов. Предположим, что переменные A и B имеют один и тот же тип X. При каких условиях можно сказать, что A = B? К сожалению, в самом языке нет средств, которые помогли бы ответить на этот вопрос. Разработка типа данных существенно зависит от того, как будут использоваться объекты этого типа. Рассмотрим следующие два определения типов в языке С для стека (stack) и множества (set):

```
struct stack
       {int TopStack;
       int Data[100]; } X, Y;

struct set
       {int NumberInSet;
       int Data[100]; } A, B;
```

Как видно из этих определений, переменные X, Y, A и B имеют структурно эквивалентные типы — целое число и массив из 100 целых чисел. Тем не менее условия, при которых выполняется X = Y и A = B, весьма различны:

- *Равенство для стеков.* Если предположить, что TopStack обозначает объект данных, расположенный на вершине стека Data, то эквивалентность X и Y имеет смысл определить следующим образом:
  1) X.TopStack = Y.TopStack;
  2) для всех I между 0 и TopStack-1 выполнено: X.Data[I] = Y.Data[I].

  Выполнение этих условий будет означать, что X и Y представляют собой эквивалентные стеки.
- *Равенство для множеств.* Если предположить, что NumberInSet обозначает количество объектов данных в множествах A и B, то эквивалентность множеств A и B следует определить так:
  1) A.NumberInSet = B.NumberInSet;
  2) множество A.Data[0]..A.Data[NumberInSet - 1] является перестановкой множества B.Data[0]..B.Data[NumberInSet - 1], так как порядок расположения элементов множества не имеет значения при определении эквивалентности.

  Разумеется, на эти определения влияет то, как мы реализуем операции над стеками push и pop (то есть добавление элемента в стек и извлечение

его оттуда) и операции над множествами insert и delete (аналогичные действия для множеств). Например, если мы реализуем добавление элемента к множеству (то есть операцию insert) таким образом, что все элементы множества будут расположены в возрастающем порядке (это можно сделать, так как элементами множества являются целые числа), то есть при добавлении каждого нового элемента заново будет происходить сортировка массива Data, то определение равенства для стеков станет применимо и для множеств.

Итак, можно сделать вывод, что в настоящее время не существует механизма формализации определения равенства для сложных структур данных. Поэтому обычной практикой при создании определяемых программистом типов данных является добавление специальных операций для установления равенства между объектами. Следовательно, если вы создаете новый тип данных, например стеки, то вам придется помимо обычных операций добавления и извлечения элемента из стека, очистки стека и возвращения верхнего элемента (push, pop, empty и top) определить еще и операцию равенства стеков (StackEquals), если, конечно, равенство между стеками используется в вашей программе.

6.4.2. Определение типов с параметрами

В некоторых языках предусмотрена возможность параметризации определения типов данных, что позволяет использовать одно и то же определение, подставляя в него различные параметры. Необходимость в параметризации возникает тогда, когда требуется определить множество похожих типов данных. Классическим примером параметров, используемых для определения типов, являются размеры массивов. Предположим, что мы хотим определить тип Section как запись. В языке Ada это будет выглядеть следующим образом:

```
type Section is
      record
         Room: integer;
         Instructor: integer;
         ClassSize: integer range 0..100;
         ClassRoll: array (1..100) of Student_ID;
      end record;
```

Отметим одно потенциальное неудобство такого определения. Здесь список студентов, входящий в объект Section, может включать в себя не более чем 100 фамилий. Если потребуется более длинный список, придется создавать новое определение типа.

Параметризованное определение типа Section позволяет ввести параметр MaxSize, который определяет максимальное количество студентов в группе:

```
type Section(MaxSize: integer) is
         record
            Room: integer;
            Instructor: integer;
            ClassSize: integer range 0..MaxSize;
            ClassRoll: array (1..MaxSize) of Student_ID;
         end record;
```

Такое определение типа позволяет задавать значение параметра MaxSize как часть объявления каждой отдельной переменной типа Section:

X: Section(100) — задает максимальный размер равным 100;
Y: Section(25) — задает максимальный размер равным 25.

В языке ML эта концепция получила дальнейшее развитие: параметром может являться и тип данных:

```
signature OrderedItem =
      sig
         type elem
         val lessthan: elem * elem -> bool
      end;
```

OrderedItem определен как тип (в ML тип определяется с помощью ключевого слова signature), который состоит из следующих компонентов:

1) параметра, имеющего тип elem;
2) функции lessthen, которая получает два элемента типа elem и возвращает булево значение.

С использованием этой сигнатуры (то есть типа) можно объявлять типы данных следующим образом:

```
(*объявление типа intitem как варианта целого типа*)
structure intitem: OrderedItem =
      struct type elem = int
      fun less(i:elem, j:elem) = (i<j)
      end;
(*объявление типа realitem как варианта вещественного типа*)
structure realitem: OrderedItem =
      struct type elem = real;
      fun less(i:elem, j:elem) = (i<j)
      end;
```

Затем мы можем использовать следующий вызов:

```
- intitem.less(5,6);
val it = true : bool
```

intitem.less — это функция less, аргументами которой являются целые числа, что указано в сигнатуре для inititem.

Реализация. Определения типов с параметрами используются при компиляции в качестве шаблонов, так же как и остальные определения типов. Единственное отличие заключается в том, что когда компилятор транслирует объявление переменной со списком параметров, следующим за именем типа, он сначала вставляет фактические значения параметров в определение типа, чтобы получить полное определение типа переменной без параметров. Используемые в определениях типов параметры влияют на реализацию выполнения программ, написанных на данном языке, только в некоторых немногочисленных случаях (например, когда подпрограмма должна получать в качестве параметра любой объект данных параметризованного типа и должна быть готова к получению параметра, возможно, иного размера при каждом вызове). В случае языка ML, в котором можно параметризировать как операции, так и типы их аргументов, фактически можно определить инкапсулированный тип. Мы обсудим вопросы реализации параметризованных типов более полно в главе 7, когда будем рассматривать объектно-ориентированные классы.

6.5. Обзор языка C++

История языка. Так же как и в случае с языком Pascal, автором которого считается Никлаус Вирт, создание языка C++ обычно приписывают одному конкретному человеку. Бьерн Страуструп на основе языка С разработал столь же эффективный язык, но с некоторыми дополнительными возможностями в области наследования объектов. Для понимания языка C++ желательно, чтобы уже был изучен язык С (приложение, раздел П.2).

В конце 70-х гг. Страуструп работал над своей докторской диссертацией в компьютерной лаборатории (Computing Laboratory) в Кембридже (Англия). В то время он использовал язык Simula (произошедший от ALGOL), в котором было введено понятие классов как объектов данных. Страуструп сделал вывод, что классы, определенные в Simula, могут служить эффективным механизмом для определения типов. Поступив на работу в компанию AT&T Bell Telephone Laboratories в США, он немедленно начал разрабатывать расширения для языка С, которые включали в себя некоторые возможности этих самых классов из Simula.

Страуструп руководствовался тем принципом, что все дополнения, привнесенные им в язык С, не должны понизить его эффективность и стать причиной замедления работы программ. Разработанные им расширения языка С, известные под названием *C with Classes* (С с классами), содержали основную структуру классов, входящую в современный C++. К 1982 г. его язык С с классами имел скромный успех в пределах AT&T Bell Telephone Laboratories, и Страуструпу, уподобившемуся гуру, приходилось решать все вопросы, связанные с поддержкой этого языка. В конце концов, эта обязанность стала для него слишком обременительной, и он пришел к выводу, что у него есть только две возможности [109]:

«1) Прекратить поддержку языка С с классами, в результате чего его пользователи будут вынуждены переключиться на какой-либо другой язык и дадут мне возможность заниматься своими делами.

2) Используя весь накопленный опыт, разработать на основе С с классами новый (более качественный) язык, который имел бы коммерческий успех. Это позволит организовать поддержку и дальнейшее развитие нового языка на коммерческом уровне и впоследствии также даст мне возможность заниматься своими делами».

Он выбрал второй вариант и усовершенствовал язык С с классами, добавив новые возможности, а также устранив некоторые несоответствия, имевшиеся в этом языке. В 1984 г. результат его труда получил новое название. Одни называли С с классами *новым С*, а другие называли стандартный С *старым*, или *простым*,С. Некоторое время использовалось название C84, а затем Рик Маскитти (Rics Mascitti) из Bell Labs предложил название C++. В этом названии используется обозначение ++ операции увеличения на единицу в языке С, которое в данном случае означает «следующий», «последователь». Название C++ многократно обыгрывалось в различных каламбурах и служило поводом для многочисленных шуток (см. врезку «Обзор языка 6.1»).

В основном разработка языка C++ была закончена к 1986 г., а в июне 1989 г. была выпущена вторая версия. В 1989 г. в Американском национальном институте стандартов (ANSI) был создан комитет X3J16, который к 1994 г. выпустил черновой стандарт языка, а через два года — его полный стандарт.

Обзор языка 6.1. С+– (произносится как «более или менее С»)

Возможности. В отличие от С++ С+– — субъективно ориентированный язык. Каждый экземпляр класса С+–, известный как субъект, имеет скрытые члены, известные как предубеждения или необъявленные предпочтения, которые являются непробиваемыми предпочтениями, которые невосприимчивы к внешним сообщениям, а также открытые члены, известные как хвастовство или амбиции. Следующие операции С перегружены в С+– так, как приведено ниже:

- `>` лучше чем;
- `<` хуже чем;
- `>>` гораздо лучше чем;
- `<<` забудьте это;
- `!` ни за что на свете;
- `==` сравнимы, две разные вещи являются одним и тем же.

С+– является строго типизированным языком, основанным на стереотипах и логике, основным постулатом которой является «Я всегда прав». Булевы переменные `истина` и `ложь` (которые в менее реалистичных языках считаются константами) дополнены такими, как `возможно` и `сомнительно`, еще менее четко определенными, чем традиционные нечеткие категории Заде. Все булевы величины могут быть объявлены с модификаторами `сильный` и `слабый`. Говорят, что слабая импликация «сохраняет отрицание» и была добавлена по требованию Министерства обороны для совместимости с будущими версиями языка Ada. Правильно построенные ложные утверждения совместимы в операции присваивания с любыми булевыми переменными. Циклы `что-если` и `почему-не` управляются специальным условным оператором `даже-если-не X тогда Y`.

С+– поддерживает сокрытие информации и, только среди дружественных классов, обмен сплетнями. Недружественные классы, заимствованные из лексикона языка Eiffel, могут взаимно уничтожить друг друга путем заключения контрактов. Заметим, что дружественные отношения между классами являются нетранзитивными, непостоянными и не-Абелевыми.

Механизмы простого и множественного наследования реализованы с использованием случайных мутаций. Правила лишения наследства (денаследование) подчиняются сложным официальным протоколам. Помимо базовых, производных, виртуальных и абстрактных классов в С+– поддерживаются *внутриутробные* классы. В некоторых локальных версиях языка разрешен вывод полигамных и побочных классов. Свободные связи между классами недопустимы, поэтому введены операции `брака` и `развода`:

```
брак(Родитель_М1,Родитель_Ж1);
//теперь можно выводить дочерние классы
//определение класса MySclass
sclass MySclass: public Родитель_М1, Родитель_Ж1
//незаконнорожденный класс!!
sclass YourSclass: public Родитель_М1, Родитель_Ж2
развод (Родитель_М1, Родитель_Ж1);
брак(Родитель_М1, Родитель_Ж2);
//теперь это по правилам
sclass YourSclass: public Родитель_М1, Родитель_Ж2
```

Правила, задающие порядок приоритетов операций, могут быть временно отменены с помощью директивы компилятора `#pragma dwim`, известной как прагма «Do What I Mean» («Делай то, что я говорю»). Язык устойчиво сопротивляется ANSIфикации. Девиз языка С+– «Каждый сам для себя стандарт».

Ссылки. Из S. Kelly-Bootle, "The Devil's AI Dictionary", *AI Expert*, (April 1991), 51. Перепечатано в S. Kelly-Bootle, "The Computer Contradictionary", MIT Press (1995). Воспроизведено с разрешения автора.

Краткий обзор языка. Язык C++ является производным от языка C. Основным отличием языка C++ является наличие классов и механизма наследования объектов класса другими классами. При разработке языка C++ руководствовались следующими тремя принципами:

1) использование классов не должно приводить к более медленному выполнению программ по сравнению с программами без классов;
2) программы на C рассматриваются как подмножество программ на C++, то есть добавление классов не должно приводить к тому, что какие-либо свойства C не будут включены в C++;
3) не должна снизиться эффективность выполнения программ.

По большей части все три цели были достигнуты. Компилятор C++ способен компилировать большинство программ на C, хотя между этими языками все же существует некоторое несоответствие. Строгая типизация C++ действительно *более строгая*, чем в C. В C++ были добавлены некоторые зарезервированные слова. Среди прочих изменений можно отметить добавление новой формы комментариев: дополнительно к C-комментариям /*...*/ был добавлен их новый формат //, означающий, что все, что следует за символами // до конца строки, является комментарием. Также расширились возможности ввода и вывода за счет добавления функций обработки потоков `cin` и `cout`. Добавлены исключения, перегрузка операций и форма общего класса под названием *шаблон*.

6.6. Рекомендуемая литература

Три классические статьи, в которых рассматриваются центральные концепции программирования — абстракция данных и сокрытие информации, — это [37], [51] и [86]. Определение и реализация различных форм абстрактных типов данных было предметом изучения и темой многочисленных статей в 70-е гг. В 1977 г. на конференции ACM [122] были описаны языки CLU, ALPHARD и EUCLID. Первым широко признанным языком с выраженной возможностью определения абстрактных типов данных стал язык Ada [57], хотя Smalltalk опередил в этом отношении Ada почти на десять лет. Но Smalltalk стал приобретать статус одного из основных языков программирования только в последнее время, о чем мы расскажем в разделе 7.2.4. В большинстве работ, посвященных созданию компиляторов, исследуются стеки и стратегия статического распределения памяти (см. ссылки в главе 3).

6.7. Задачи и упражнения

1. Пусть у нас имеется связанное представление вектора `V`, как на рис. 6.1. Напишите алгоритм для определения местоположения компонента `V[N]`. Предполагается, что дескриптор имеет адрес α, его поле с указателем на первый компонент вектора в блоке дескриптора имеет сдвиг `j` и каждый компонент хранится в отдельном блоке памяти. В каждом блоке сдвиг поля указателя на следующий компонент равен `k`.

2. Предположим, что объявление вектора V задано с использованием некоторого перечисляемого типа в качестве набора индексов. Например,

```
ClassType = (Fresh, Soph, Junior, Senior, Graduate);
V: array [ClassType] of real;
```

✦ Предложите подходящий способ представления в памяти вектора V (включая дескриптор) и выведите формулу доступа для вычисления местоположения компонента V[i].

✦ Покажите, как изменится способ представления и формула доступа, если V будет объявлен как

```
V: array [Junior..Graduate] of real;
```

3. Приведите формулу доступа для вычисления местоположения элемента матрицы A[I, J], развернутой по столбцам, объявление которой выглядит следующим образом:

A: array [LB_1.. UB_1, LB_2.. UB_2]

4. Многие вычисления с использованием матриц включают в себя последовательную обработку всех элементов какого-либо столбца или строки. Обычно эта обработка реализована в виде цикла по одному из индексов I или J, в котором элементы матрицы A[I, J] обрабатываются последовательно по мере увеличения индекса на единицу. В таких случаях независимое вычисление *l*-значения для каждого элемента A[I, J] в каждом цикле очень неэффективно. Можно гораздо проще вычислить *l*-значение элемента A[I, J] — через *l*-значение элемента A[I - 1, J] (в предположении, что меняется индекс I). Напишите формулу для такого рекурсивного вычисления адреса элементов матрицы. Обобщите эту формулу на случай массива с произвольной размерностью для цикла, в котором изменяется произвольный индекс этого массива.

5. В языке SIMSCRIPT многомерные однородные массивы представлены как векторы указателей, которые указывают на другие векторы, состоящие из указателей и т. д. Количество уровней этой структуры векторов совпадает с размерностью исходного массива. Числовая матрица размером 3 × 4 представлена вектором с тремя указателями, каждый из которых указывает на вектор, состоящий из четырех элементов. Для такого представления предложите алгоритм доступа к элементу A[I, J]. Сравните относительную эффективность использования памяти и доступа к элементам массива между этим способом представления и обычным последовательным представлением массива. Рассмотрите этот вопрос как для матриц, так и для массивов более высоких размерностей.

6. Для многомерных массивов:

✦ предложите способ создания дескриптора для векторных сечений, если имеется дескриптор всего массива;

✦ сконструируйте дескриптор времени выполнения для двухмерных сечений (плоскостей в трехмерном массиве), который позволял бы использовать такую же формулу доступа, как и для обычных матриц;

✦ в случае произвольного двухмерного квадратного массива (то есть массива A[n, n]) разработайте дескриптор для вектора, представляющего собой главную диагональ массива (то есть для вектора A[i, i]).

7. Поскольку большая часть аппаратных средств компьютеров не обеспечивает непосредственную поддержку проверки соответствия значений индексов массива заданному диапазону при доступе к его компонентам, стоимость этой проверки, проводимой каждый раз при обращении к какому-либо компоненту массива, может оказаться достаточно высокой как в отношении времени выполнения, так и в отношении памяти, требуемой для хранения дополнительного кода. Выберите какой-либо известный вам язык программирования и определите для этого языка относительную стоимость (в отношении времени выполнения дополнительного кода и необходимой памяти) проверки индексов при доступе к компонентам двухмерной матрицы `A[I, J]`. При возможности распечатайте листинг объектного кода, сгенерированного компилятором языка, в котором осуществляется проверка индекса на принадлежность диапазону, для какой-либо простой программы, выполняющей доступ к элементам некоторого массива. После его анализа определите, какие инструкции сгенерировал компилятор для проверки соответствия значений индекса объявленному диапазону и какое время было затрачено на их выполнение в случаях с проверкой и без проверки.
8. Рассмотрим используемое в языке Pascal представление множества через битовые строки. Предположим, что максимальное количество элементов в множестве должно быть меньше, чем длина слова, обусловленная аппаратной частью компьютера. Предложите алгоритмы реализации операций объединения, пересечения и разности множеств и определения принадлежности к множеству (операция `in`) на основе следующих встроенных в аппаратуру простейших операций: логическое И, логическое ИЛИ и логическое дополнение, применяемые к целым словам.
9. При использовании хэш-кодирования для представления множеств в памяти разрешение конфликтных ситуаций реализуется с помощью методик как повторного хэширования, так и последовательного сканирования. Если разрешено удаление элементов из множества, возникают некоторые сложности при использовании обеих методик. Объясните эти сложности.
10. Выберите какой-нибудь язык программирования, в котором определен структурный тип данных. Перечислите *атрибуты* объектов данных этого типа. Перечислите *операции*, определенные в языке для объектов этого типа данных. Определите представление в памяти объектов данных этого типа в реализации языка на вашем компьютере. Имеется ли у этих объектов дескриптор во время выполнения программы? Какие атрибуты хранятся в этом дескрипторе? При использовании этого представления требуется ли при выполнении каких-либо операций выделение дополнительного объема или, наоборот, освобождение памяти?
11. Выберите какой-нибудь язык программирования и структурный тип данных в нем. Определите, какие имеются *операции* выборки отдельных элементов или подструктур объекта данных этого типа. Для каждой операции выборки (или класса таких операций) ответьте на следующие вопросы:
    а) может ли существование выбранного компонента быть определено статически (то есть во время компиляции);

б) может ли тип данных выбранного компонента быть определен статически? Если существование или тип данных не могут быть определены статически, то как именно происходит динамическая проверка во время выполнения программы — является ли эта проверка обязанностью самого программиста или осуществляется автоматически, то есть предусмотрена реализацией языка?

12. Допустим, что у вас имеется модифицированная версия языка Pascal, в которой имена полей в записях (не вариантных) могут быть целыми числами, а при выборке полей можно вычислять целые значения имен полей, так что, например, для выборки компонента записи можно использовать выражение `R.(I + 2)`.
    - ✦ Объясните, почему в такой версии представление записей в памяти, описанное в разделе 6.1.6, перестанет быть адекватным?
    - ✦ Модифицируйте это представление так, чтобы оно могло функционировать в новой версии, и предложите формулу доступа (или алгоритм), который можно было бы использовать в этом новом представлении для выбора компонента `R.K`, где `K` — это вычисленное значение.
13. Повторите задание 12, но для записей с вариантными полями.
14. Для языка, в котором допускаются вариантные записи без полей тегов (свободное объединение), как, например, в Pascal, напишите процедуру

    ```
    procedure GIGO(I: integer: var R: real:)
    ```

    которая использует запись с двумя вариантами. Единственным назначением этой процедуры является попытка сбить с толку систему проверки типов. Процедура `GIGO` получает параметр I целочисленного типа и возвращает в качестве результата ту же комбинацию битов, но как вещественное число `R`, при этом не осуществляя фактического преобразования целого значения I к вещественному числу.
15. Выберите какой-либо знакомый вам язык и постарайтесь выяснить, насколько эффективна инкапсуляция каждого из элементарных типов данных. Для этого нужно написать несколько тестовых программ, с помощью которых, может быть, удастся определить некоторые свойства представления объекта данных каждого типа в памяти. Чем меньше таких свойств можно определить, тем более эффективна инкапсуляция.
    - ✦ *Массивы.* Можете ли вы определить, какой способ развертывания использован в данном языке — по столбцам или по строкам? Можно ли определить, имеется ли дескриптор времени выполнения и каковы его формат и содержание?
    - ✦ *Записи.* Можете ли вы определить, в каком порядке хранятся компоненты записи? Используется ли упакованная форма хранения или расположение каждого компонента привязано к границам адресуемых единиц памяти?
16. Выберите какой-либо язык программирования и определите в нем абстрактный тип данных *стек целых чисел* и операции `push` и `pop`, которые добавляют и извлекают элементы стека. Представьте, что вы входите в группу программистов, которая разрабатывает какую-нибудь большую программу, и в этой

программе часто используются стеки целых чисел. Объясните, как наиболее эффективно реализовать разработанный вами абстрактный тип данных (используя комбинацию ограничений языка и соглашений по программированию, которым должны следовать все разработчики группы при работе со стеками), чтобы представление стеков в памяти было полностью скрыто и манипулировать им можно было бы только при помощи операций push и pop.

17. Для какой-либо недавно написанной вами программы приведите полную спецификацию параметров и результатов каждой подпрограммы. Не забудьте также и о неявных параметрах и результатах. Есть ли в вашей программе какие-нибудь подпрограммы:
    1) зависящие от предыстории их вызовов;
    2) неопределенные для некоторых параметров из области их задания? Для каждой подпрограммы приведите хотя бы один пример сокрытия информации.
18. Подпрограмма, которая компилируется в отдельную запись активации и отдельный сегмент кода, иногда называется *повторно входимой*, так как во время работы программы ее можно вызвать второй раз, до завершения ее первой активации. Таким образом, может существовать несколько активаций, совместно использующих один и тот же сегмент кода. Для какого-либо знакомого вам языка определите, являются ли его подпрограммы повторно входимыми. Какое средство (или средства) языка позволяет запускать вторую активацию до того, как закончилась первая?
19. Для какого-либо знакомого вам языка определите, как в нем определена эквивалентность типов — через эквивалентность имен или структурную эквивалентность? Рассмотрите каждый тип данных отдельно (поскольку способ определения эквивалентности для разных типов может быть разным) и точно объясните, когда две переменные соответствующего типа считаются имеющими одинаковый тип и когда переменная, представляющая фактический параметр, и формальный параметр подпрограммы считаются имеющими одинаковый тип? Различаются ли способы определения эквивалентности для простейших типов данных и для определяемых программистом?
20. Предложите три различных способа определения эквивалентности типа для *записей* и два способа для *векторов*, основываясь на понятии структурной эквивалентности.
21. Функция ord определена следующим образом: ее единственным аргументом является элемент некоторого перечисления, а результатом является порядковый номер этого элемента в перечислении (0 соответствует первому элементу перечисления, 1 — второму и т. д.). Однако фактически эта функция не выполняет никаких действий, поскольку элементы перечисления во время выполнения программы и так представлены целыми числами, соответствующими расположению элементов в последовательности (перечислении). Например, если объект, принадлежащий типу *перечисление*, задан как

    ```
    Class = (Fresh, Soph, Junior, Senior)
    ```

    и переменная X относится к типу Class, то операция присваивания X := Senior фактически присваивает переменной X значение 3 и соответственно резуль-

татом вычисления функции ord(X) будет целое число 3. Таким образом, функция ord() получает значение 3 и возвращает значение 3, то есть фактически не делает ничего полезного (и реализация языка Pascal совершенно безболезненно может изъять эту функцию из исполняемого объектного кода). Тем не менее эта функция играет определенную роль, но эффект ее выполнения проявляется во время компиляции.

- Объясните, каково назначение функции ord. Почему Pascal предоставляет эту функцию, когда она совсем не имеет никакого эффекта во время выполнения программы?
- Для переменной X, определенной выше, объясните действия, производимые во время *выполнения* следующих двух фрагментов:

```
if X = Senior then ...
```

и

```
if ord(X) = 3 then ...
```

- Объясните действия, предпринимаемые во время *компиляции* для проверки типов в двух фрагментах, приведенных выше, и в третьем фрагменте:

```
if X = 3 then ...
```

22. Объекты cin и cout ввода и вывода потока данных в C++ реализованы как экземпляры класса stream. Фактически считывание и запись данных происходят за счет перегрузки операций сдвига << и >>. Предложите реализацию этих функций с использованием стандартных функций ввода-вывода языка C.

23. Разработайте в C++ класс rational (класс рациональных чисел), объекты которого представляют собой частное от деления двух целых чисел a и b, то есть a/b. Перегрузите операции C++ таким образом, чтобы операторы типа

```
a = b + c
```

могли использоваться для a, b и c, принадлежащих как к целому типу, так и к классу rational.

- Разработайте класс рациональных чисел для операций +, - и *. Включите функцию print(x) для рационального аргумента x, которая печатала бы значение x как y/z. Если первый аргумент рациональный, то вычисление функции очевидно. Что будет, если первый аргумент целочисленный? Как бы вы разработали эти операции, если бы оба аргумента принадлежали к целому типу? (Для решения этой задачи вы должны воспользоваться дружественными функциями (friend).)
- Попробуйте расширить класс, включив в него операцию деления /. Какие при этом возникают проблемы и как можно их решить? Помните, что результат деления a/b также должен быть рациональным числом.

24. Предположим, что в языке BL определена структура данных *стек* и три операции:

NewTop(S, E) — добавление элемента E в вершину стека S;

PopTop(S) — удаление верхнего элемента стека S;

GetTop(S) — возвращение указателя на текущий верхний элемент стека S.

Что неправильно в разработке этих трех операций? Как следует их переопределить, чтобы исправить положение?

25. Список свойств может быть представлен как множество (а не как список), поскольку доступ к его элементам осуществляется произвольным образом при помощи индекса (имени свойства), а не последовательно, что более характерно для списков. Разработайте представление в памяти списков свойств, используя предложенную в разделе 6.1.7 методику.

26. В языке SNOBOL4 список свойств, или *таблица* (в терминологии этого языка), создается при помощи оператора, подобного следующему:

```
X = TABLE(50,20)
```

Таблицы хранятся в памяти с использованием смешанного последовательного и связанного представлений. В памяти выделяется начальный блок, достаточно большой для хранения 50 пар «индекс–значение», и указатель на этот блок присваивается переменной X. Пары «индекс–значение» помещаются в таблицу с помощью операций присваивания вида

```
X[Age] = 52
```

который заносит пару (Age. 52) в таблицу, если в ней индекс Age еще не существует. Если же этот индекс уже присутствует в таблице, то соответствующее ему значение меняется на 52. Когда в начальный блок будет занесено 50 пар «индекс–значение», то в памяти будет выделен новый блок вместимостью до 20 пар (второй параметр при вызове функции TABLE), который будет связан с первым блоком при помощи указателя. Новые пары теперь будут заноситься в этот новый блок, пока он не заполнится, что приведет к выделению еще одного блока снова вместимостью до 20 пар. Удаление пар из блока не допускается. Разработайте детальную структуру представления таких таблиц в памяти, включая дескриптор времени выполнения, а затем предложите алгоритм для реализации приведенной выше операции присваивания в случае произвольной таблицы.

27. В синтаксическом представлении списка с использованием обычной *списочной нотации* (как последовательности элементов, заключенных в круглые скобки) завершающий cdr указатель на nil подразумевается неявным образом (например, записывают как (A B C), а подразумевают (A B C nil)). Иногда требуется, чтобы последний элемент списка имел cdr указатель на атом, отличный от nil. В этом случае можно использовать альтернативную форму записи — *точечную нотацию*. В точечной нотации каждый элемент списка записывается в виде пары подэлементов, представляющих собой car и cdr элемента. Подэлементы заключаются в круглые скобки и разделяются точкой. Например, список (A) в списочной нотации преобразуется в (A.nil) в точечной нотации, (A B) запишется как (A.(B.nil)) и ((A B) C) — как ((A.(B.nil)).(C.nil)). Теперь пара из элементов A и B, которая не может быть записана в списочной нотации, может быть записана в точечной нотации как (A.B). Перепишите в точечной нотации следующие списки:

а) (A (B C));

б) (((A)) B (C D)).

28. Списки свойств атомов, которые содержат свойства, отличные от печатаемого имени атома, никогда не могут быть удалены утилитой сборки мусора, даже если они полностью недоступны из активной списковой структуры во время сборки мусора. Объясните почему.

29. *Самовоспроизводящаяся функция.* Эквивалентность представления программы и данных в языке LISP позволяет достаточно легко писать довольно сложные программы, реализация которых на других языках была бы значительно труднее. Некоторые из этих программ имеют статус *классических задач* LISP, среди которых типичной является задача самовоспроизводящейся функции. Напишите на языке LISP функцию SRF, значением которой было бы ее собственное определение. Функция SRF не получает никаких параметров. Если SRF определена как

    ```
    (defun SRF ()body)
    ```

    то результатом вызова SRF является списочная структура (defun SRF ()body). Функция SRF должна собирать список, являющийся результатом ее вычисления, по частям. Она не может иметь доступа к списку свойств атома SRF для того, чтобы получить список, являющийся ее определением.

30. Рассмотрим следующую программу на языке ML:

    ```
    datatype digit = zero | one | two | three | four | five
          | six | seven | eight | nine;
    datatype number = sdigit of digit | node of number * number;
    fun digitv(one) = 1 | digit(two) = 2 | digitv(three) = 3
          | digitv(four) = 4 | digitv(five) = 5 | digitv(six) = 6
          | digitv(seven) = 7 | digitv(eight) = 8 | digitv(nine) = 9
          | digitv(zero)=0;
    fun value(node(x,y)) = 10 * value(x) + value(y)
          | value(sdigit(x)) = digitv(x);
    ```

    - Постройте трассировку выполнения каждой из приведенных ниже функций. Какое значение будет напечатано в результате выполнения каждой из них?

      1) `value(sdigit(seven));`

      2) `value(node(sdigit(nine), sdigit(three)));`

      3) `value(node(sdigit(three), node(sdigit(one), sdigit(four)))).`

    - Изобразите схематически структуру данных для каждого выражения из предыдущего пункта.

Глава 7. Наследование

В разделе 6.2 мы обсуждали концепцию инкапсулированных типов данных как средство разработки программ, которое позволяет создавать новые типы данных с операциями, выполняющимися только над объектами этих новых типов. Например, для создания списков студентов, посещающих различные курсы лекций в университете, можно создать новые типы данных — section (группа) и student (фамилия студента). Для занесения фамилий студентов в списки групп определенного курса можно разработать операцию AddToSection с сигнатурой student × section → section.

В программе

```
typedef { definition } section;
typedef { definition } student;
section NewClass;
student NewStudent;
AddToSection(newStudent, NewClass);
```

подробности реализации объектов типа student и section скрыты в подпрограмме AddToSection. Программист может рассматривать эти два новых типа как обычные элементарные типы данных, а подпрограмму AddToSection — как элементарную встроенную в язык функцию. Знание фактической структуры этих типов данных потребуется только разработчику подпрограммы AddToSection. Хотя подобная методика может применяться для написания программ на любом языке, хотелось бы максимально упростить ее использование и по возможности избежать связанных с ее применением ошибок. Лучше иметь в распоряжении такой язык программирования, который помогает осуществлять инкапсуляцию данных, а не полагаться на то, что программисты не будут ошибаться.

Сначала мы опишем механизмы автоматической инкапсуляции данных, такие, например, как конструкция package (пакет) в языке Ada. Затем мы расширим эту концепцию таким образом, чтобы операции, применяемые к инкапсулированным объектам данных, можно было выводить автоматически, используя так называемую концепцию *наследования*. Выводимые таким образом операции называются *методами*. Логическим развитием этой концепции является полный полиморфизм операций, который также обсуждается в этой главе.

7.1. Повторное рассмотрение абстрактных типов данных

Напомним, что в разделе 6.2 мы определили *абстрактный тип данных* как новый тип данных, определяемый программистом и включающий:

1) определяемый программистом тип данных;
2) набор абстрактных операций над объектами этого типа;
3) инкапсуляцию объектов этого типа таким образом, что пользователь нового типа не может манипулировать этими объектами, иначе как только с помощью определенных при разработке типа абстрактных операций.

Абстракция данных, то есть конструирование новых типов данных и операций над объектами этих типов, как было сказано выше, является одной из фундаментальных концепций программирования. В тех языках, которые плохо обеспечивают прямую поддержку абстракции данных сверх обычного механизма определения подпрограмм, программисту все же предоставлена возможность создавать и использовать абстрактные типы данных, несмотря на то что такое понятие в языке не представлено. Поэтому программист должен использовать соглашения по кодированию, организуя программу таким образом, чтобы достичь эффекта использования абстракции типов данных. Однако без поддержки определения абстрактных типов данных в языке *инкапсуляция* нового типа данных невозможна. Так, например, если нарушены соглашения по кодированию (сознательно или случайно), то реализация языка не сможет обнаружить их нарушения. В таких языках, как C, FORTRAN и Pascal-подобные, определенные программистом абстрактные типы данных часто появляются в виде специальных библиотек подпрограмм. Ada, Java и C++ относятся к тем немногим широко распространенным языкам, в которых имеются специальные средства для поддержки абстракции данных.

Способ определения типов, подобный тому, что используется в языке C, упрощает объявление новых переменных данного типа, так как для этого требуется только имя этого типа. Однако внутренняя структура объектов данных этого типа не инкапсулирована. Любая подпрограмма, в которой можно объявить переменную как принадлежащую к новому типу данных, также позволяет получить доступ к компонентам представления этого типа. Таким образом, она может игнорировать операции, специально разработанные для манипулирования с объектами данных этого типа, и вместо этого непосредственно обращаться к их компонентам. Основная цель инкапсуляции определения абстрактного типа данных — сделать такой доступ невозможным, чтобы только подпрограммы, которые «знают», как представлены объекты данных этого типа, и являлись операциями, определенными как часть абстрактного типа.

Из тех языков, которые описаны в этой книге, только Ada, C++ и Smalltalk поддерживают инкапсуляцию таких определений абстрактных типов данных. В языке Ada определение абстрактного типа данных представляет одну из форм *пакета* (конструкция `package`). Пакет, определяющий абстрактный тип данных `SectionType`, может быть определен в виде, представленном в листинге 7.1. Объявление `is private` для типа `Section` означает, что внутренняя структура объектов данных `Section` недоступна из подпрограмм, использующих этот пакет. Фактические детали этого

типа даны в конце пакета в компоненте private. Только подпрограммы, определенные в самом пакете, имеют доступ к этим закрытым данным. Так, например, процедуры AssignStudent и CreateSection могут обращаться к массиву ClassRoll, который является одним из компонентов типа Section. Но никакие другие процедуры, внешние по отношению к пакету, не имеют доступа к этому компоненту, хотя могут объявлять переменные как объекты типа Section (и указывать значение параметра, задающего максимальное количество студентов в группе).

Реализация. Определение абстрактных типов данных в виде пакетов Ada не затрагивает никаких новых идей, относящихся к их реализации. Пакет обеспечивает инкапсуляцию для множества определений типов и подпрограмм. Таким образом, его основное достижение заключается в ограничении видимости имен, объявленных в пакете, чтобы пользователи абстрактного типа данных не смогли получить доступа ко внутренним элементам определения. Если же компилятор определяет, что некоторая процедура имеет доступ к определению типа, то применяются алгоритмы размещения объектов данных в памяти и доступа к ним, описанные в предыдущих главах.

Хотя, как только что было сказано, реализация пакетов языка Ada не использует новые идеи реализации, все же стоит рассмотреть концепцию пакетов более подробно. Каждый пакет состоит из двух частей: спецификации и реализации. Как видно из листинга 7.1, спецификация пакета SectionType определяет все данные, типы и процедуры, которые известны и видимы для подпрограмм, определенных в других пакетах. Реализация процедур, объявленных в спецификации пакета, задается в компоненте package body (тело пакета). В тело пакета могут входить также дополнительные объекты и типы данных, которые не должны быть видны из других пакетов. В данном случае процедура ScheduleRoom вызывается только из процедур AssignStudent и CreateSection, и ее имя просто неизвестно вне данного пакета.

Листинг 7.1. Определение абстрактного типа данных Section в языке Ada

```
package SectionType is
        type StudentID is integer;
        type Section(MaxSize: integer) is private;
        procedure AssignStudent(Sect: in out Section;
           Stud in StudentID);
        procedure CreateSection(Sect: in out Section;
           Instr in integer;
           Room in integer);
        private
           type Section(MaxSize: integer) is
              record
                 Room:integer;
                 Instructor: integer;
                 ClassSize: integer range 0..MaxSize := 0;
                 ClassRoll: array (1..MaxSize) of StudentID;
              end record;
        end;
package body SectionType is
        procedure AssignStudent(j) is
              -  Операторы, вставляющие объект student в ClassRoll
           end;
        procedure CreateSection(j) is
```

продолжение ⇨

Листинг 7.1 (*продолжение*)

```
        - Операторы, инициализирующие компоненты записи Section
    end;
procedure SheduleRoom(j) is
        - Операторы, определяющие расписание лекций в аудитории
    end;
end;
```

Необходимость указания имени процедуры в спецификации пакета объясняется достаточно просто: компилятору нужно знать сигнатуру процедуры, если она вызывается из другого пакета, а типы формальных параметров каждой процедуры задаются именно в спецификации пакета. Например, при вызове процедуры AssignStudent из подпрограммы какого-либо другого пакета необходима информация о том, что у данной процедуры имеются два фактических параметра: один типа Section (являющийся одновременно входным и выходным in out) и второй параметр типа StudentID (являющийся только входным in).

Но почему же определения закрытых данных, которые используются только в теле данного пакета и неизвестны за его пределами, также помещаются в его спецификации? Казалось бы, проще ограничиться помещением их в тело пакета. Чтобы понять, почему в языке Ada пакеты определены именно таким образом, нужно рассмотреть два типичных способа реализации абстрактных типов данных.

На рис. 7.1 представлены две модели реализации инкапсулированных объектов данных. На рис. 7.1, *а* схематически изображен пример *непрямой инкапсуляции*. В этом случае структура абстрактного типа данных определяется спецификацией пакета A. Фактическое местоположение объекта P задается и контролируется в записи активации для пакета A. В записи активации пакета B, который объявляет и использует объект P, содержится указатель на фактическое местоположение объекта.

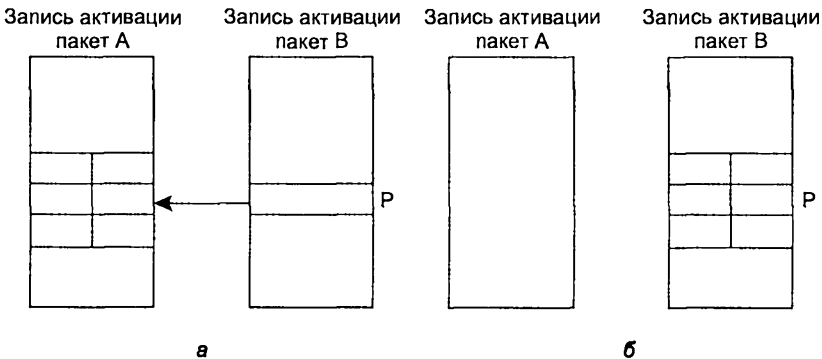


Рис. 7.1. Две модели реализации абстрактных типов данных: а — непрямая инкапсуляция объекта P; *б* — прямая инкапсуляция объекта P

Альтернативная реализация, называемая *прямой инкапсуляцией*, схематически изображена на рис. 7.1, *б*. Как и в случае непрямой инкапсуляции, структура объекта данных абстрактного типа определена в спецификации для пакета A. Но теперь фактическое местоположение объекта P контролируется записью активации для пакета B.

В чем же разница между этими двумя способами? В случае непрямой инкапсуляции реализация абстрактного типа данных действительно не зависит от его использования. Если потребуется изменить внутреннюю структуру объекта P, то придется менять только пакет A. Пакету B только необходимо «знать», что объект P является просто указателем, а формат данных, на который указывает P, для пакета B не имеет значения. Способ непрямой инкапсуляции хорош для больших программ, содержащих тысячи модулей, так как отпадает необходимость заново компилировать каждый модуль, где используется объект, определение которого поменялось; за счет этого можно значительно сэкономить время.

Но, с другой стороны, при использовании этого способа дополнительное время тратится на доступ к инкапсулированным объектам при выполнении программы. Доступ к объекту P каждый раз подразумевает обращение к указателю на этот объект. Хотя само по себе это действие не связано со значительными затратами времени, многократное обращение к объекту через указатель может оказаться довольно дорогостоящим.

Прямая инкапсуляция характеризуется прямо противоположными свойствами. В данном случае объект данных P хранится в записи активации пакета B. Доступ к компонентам объекта P теперь может быть осуществлен быстрее, поскольку в пределах локальной записи активации можно применить стандартную технику доступа к объектам данных по формуле базовый адрес + сдвиг; в данном случае не требуется никаких промежуточных указателей. Но если представление объекта данных меняется, то все пакеты (например, пакет B), в которых этот объект используется, должны быть заново скомпилированы. Это увеличивает затраты на компиляцию, но ускоряет выполнение программы.

В языке Ada используется прямая модель инкапсуляции, которая обеспечивает максимальную эффективность выполнения программы. При трансляции используемого в пакете объекта абстрактного типа (например, объекта P в пакете B на рис. 7.1, *б*) требуется информация о деталях представления объекта. Этим и объясняется необходимость раздела private в спецификации пакета.

Заметим, однако, что и прямая, и непрямая модели инкапсуляции могут использоваться в любой программе, поддерживающей инкапсуляцию, независимо от модели, реализованной как часть программной архитектуры используемого языка программирования. Хотя в реализации самого языка Ada предпочтение отдано модели прямой инкапсуляции, которая и используется в нем по умолчанию, непрямая инкапсуляция может быть реализована самим программистом. В листингах 7.2 и 7.3 приведены фрагменты пакетов Ada, демонстрирующие применение обеих стратегий реализации инкапсуляции. Листинг 7.2 соответствует модели, схематически изображенной на рис. 7.1, *а* (access — переменная-указатель в языке Ada), а листинг 7.3 — модели, представленной на рис. 7.1, *б*.

Листинг 7.2. Пример непрямой инкапсуляции данных в языке Ada

```
package A is
    type MyStack is private;
    procedure NewStack(S: out MyStack);
    ...
    private
    type MyStackRep;
```

продолжение ⇨

Листинг 7.2 (*продолжение*)

```
            – Скрытые детали представления MyStack
  type Mystack is access MyStackRep;
            – В пакете В содержится только указатель на стек
end;
```

Листинг 7.3. Пример прямой инкапсуляции данных в языке Ada

```
package A is
   type MyStack is private;
   procedure NewStack(S: out MyStack);
   ...
   private
   type MyStack is record;
      Top: integer;
      A: array (1..100) of integer;
      end record;
            – В пакете В содержится информация о структуре стека
end;
```

Некоторая разновидность прямой инкапсуляции, пример которой приведен в листинге 7.3, дается следующим определением типа:

```
package A is
      type MyStack is record
      Top: integer;
      A: array (1..100) of integer;
      end record;
```

В данном случае организация записи активации совпадает с моделью прямой инкапсуляции; тем не менее все имена переменных из пакета В видны. Этот механизм характерен для языков типа Pascal, в которых типы данных реализованы без инкапсуляции.

Общие абстрактные типы данных

Элементарные типы данных, встроенные в язык, часто позволяют программисту объявлять базовый тип для нового класса объектов данных и затем уточнять некоторые атрибуты последних. Это простая форма полиморфизма, который более подробно обсуждается в разделе 7.3. Например, в языке Java имеется базовый тип данных *массив*, для которого определены некоторые элементарные операции, например операция индексации. Однако в определении класса Vect (в нижеследующем примере) в Java задается определение массива Vect.X, который содержит объекты целого типа:

```
class Vect {int [] X = new int [10]}
```

Здесь X является массивом из десяти компонентов целого типа, доступ к которым осуществляется при помощи индексов 0, 1, 2, ..., 9. Можно написать подпрограммы, которые будут манипулировать объектами класса Vect, но те операции, которые были определены для базового типа данных (то есть в данном случае для массивов), будут также оставаться доступными.

При определении абстрактных типов данных желательно также использовать подобную структуру. Например, в языке Ada определение нового абстрактного типа данных *стек* с операциями push и pop, которые вставляют и удаляют элементы стека, можно реализовать в форме пакета, текст которого приведен в листинге 7.4.

Отметим возникающую здесь проблему: определение типа элемента стека является частью общего определения стека, поэтому наше определение является определением только стека целых чисел. Для стека вещественных чисел или объектов типа Section требуется отдельное определение в другом пакете, хотя представление стека и операций push и pop может быть определено точно таким же образом.

Листинг 7.4. Определение стека целых чисел как абстрактного типа данных в языке Ada

```
package IntStackType is
      type Stack(Size: Positive) is private;
      procedure Push(I: in integer; S: in out Stack);
      procedure Pop(I: out integer; S: in out Stack);
private
      type Stack(Size: Positive) is record
         StkStorage: array (1..Size) of integer;
         Top: integer range 0..Size := 0;
         end record;
end IntStackType;
package body IntStackType is
      procedure Push(I: in integer; S: in out Stack)is
         begin
            -  Тело процедуры Push
         end;
      procedure Pop(I: out integer; S: in out Stack)is
         begin
            -  Тело процедуры Pop
         end;
end IntStackType;
```

Определение общих абстрактных типов данных позволяет такой атрибут типа задавать отдельно, для того чтобы дать определение одного базового типа с атрибутами в виде параметров с последующим созданием нескольких специализированных типов, выведенных из одного и того же базового типа. Эта структура похожа на структуру определения типа с параметрами, за исключением того, что в ней параметры могут влиять не только на определение самого типа, но и на определение операций в определении абстрактного типа, а параметрами могут быть не только числовые значения, но и имена типов. Пакет Ada, приведенный в листинге 7.5, дает пример такого определения общего типа для типа *общий стек*, в котором и тип хранимого в стеке элемента, и максимальный размер стека определены как параметры этого общего типа.

Листинг 7.5. Абстракция общего стека в языке Ada

```
generic
      type Elem is private;
package AnyStackType is
      type Stack(Size: Pozitive) is private;
      procedure Push(I: in Elem; S: in out Stack);
      procedure Pop(I: out Elem; S: in out Stack);
private
      type Stack(Size: Pozitive) is record
         StkStorage: array (1.. Size) of Elem;
         Top: integer range 0..Size := 0;
         end record;
```

продолжение ⇨

Листинг 7.5 (*продолжение*)

```
end AnyStackType:
package body AnyStackType is
       procedure Push(I: in Elem; S: in out Stack)is
          begin
             -  Тело процедуры Push
          end;
       procedure Pop(I: out Elem: S: in out Stack)is
          begin
             -  Тело процедуры Pop
          end;
end AnyStackType;
```

Порождение конкретного типа из определения общего типа. Определение общего пакета представляет собой *шаблон*, который можно использовать для создания определения конкретного типа данных. Процесс конструирования конкретного определения типа из общего определения для заданного набора параметров называется *порождением* (instantiation), или *созданием*, типа. Например, из общего определения стека, приведенного в листинге 7.5, можно создать определение стека целых чисел, эквивалентного тому, который определен в пакете IntStackType (см. листинг 7.4), при помощи объявления

```
package IntStackType is
       new AnyStackType(elem => integer);
```

Стек, содержащий данные типа Section, можно определить при помощи следующего порождения:

```
package SetStackType is
       new AnyStackType(elem => Section);
```

Далее можно определять стеки целых чисел, имеющие различные максимальные размеры:

```
Stkl: IntStackType.Stack(100);
NewStk: IntStackType.Stak(20);
```

Аналогично можно задавать размеры стеков, содержащих объекты типа Section:

```
SecStack: SetStackType.Stack(10);
```

Отметим, что общий тип AnyStackType можно использовать для порождения конкретных типов много раз с различными значениями параметров, и каждый раз мы будем получать другое определение типа с именем Stack в соответствующем пакете. Таким образом, при ссылке на тип Stack в каком-либо объявлении программы может возникнуть неоднозначность. Для ее разрешения язык Ada требует, чтобы имя пакета предшествовало имени типа, например IntStackType.Stack или SetStackType.Stack.

В C++ имеется аналогичное понятие *шаблон* (template), который используется для определения общих классов, например:

```
template <class имя_типа> class имя_класса определение_класса
```

Здесь задается неограниченное множество определений классов для всех параметров *имя_типа*.

Реализация. Реализация общих типов данных, в принципе, осуществляется непосредственно. Когда в программе создается определение типа, требуется передать в пакет с определением общего типа конкретные значения параметров. Компилятор использует определение общего типа как шаблон, вставляя в него задан-

ные значения параметров и компилируя затем определение так же, как если бы оно было обычным пакетным определением без параметров. Этот способ можно сравнить с макроопределениями #define в языке С. В процессе выполнения программы появляются только объекты данных и подпрограммы; определение пакета служит в первую очередь для того, чтобы ограничить область видимости этих объектов данных и подпрограмм. Сами пакеты не являются частью структуры исполняемого кода программы.

Если в программе на основе определения общего порождается много конкретных типов (как могло бы случиться, если бы пакет с определениями общих типов поставлялся в виде библиотеки), то такая непосредственная реализация может оказаться слишком неэффективной, так как каждое порождение нового конкретного типа требует включения в программу копии всего пакета вместе со всеми определенными в нем подпрограммами, которые затем должны быть заново перекомпилированы. Было бы лучше, если бы реализация позволяла избегать создания новой копии каждой подпрограммы, а также полной перекомпиляции всего пакета. Эта тема подробнее обсуждается в разделе 7.3, посвященном полиморфизму.

7.2. Наследование

Информация, известная в одной части программы, часто должна использоваться в другой ее части. Например, использование фактических параметров вызывающей подпрограммы в качестве формальных параметров некоторой вызываемой подпрограммы представляет собой механизм передачи значений фактических параметров вызывающей подпрограммы в вызываемую подпрограмму. В данном случае эта связь между различными частями программы осуществляется явным образом — ее формирует конкретный вызов подпрограммы.

Тем не менее довольно часто информация от одного компонента программы к другому передается неявным образом. Такая передача информации называется *наследованием*. Наследование можно определить как получение каким-либо компонентом подпрограммы свойств или характеристик некоторого другого компонента в соответствии со специальными отношениями, существующими между этими компонентами. Концепция наследования часто применяется при разработке языков программирования.

Раннюю форму наследования можно обнаружить в правилах определения области видимости для структурированных блоков данных. Имена, используемые во внутреннем блоке, могут быть унаследованы из внешнего блока. Рассмотрим, например, следующий блок данных:

```
{int i, j;
      {float j, k;
      k = i + j;}
}
```

В операторе присваивания k = i + j и k, и j являются локальными вещественными переменными, объявленными во внутреннем блоке. Но переменная i унаследована из внешнего блока, а наследование объявления int j блокируется во

внутреннем блоке благодаря переопределению типа переменной j как вещественного.

Хотя правила определения области видимости в языках Ada, C и Pascal являются разновидностью наследования, этот термин чаще используется для ссылки на передачу данных и функций между независимыми модулями программы. Примером этого является конструкция class в языке C++. Если между классами A и B установлено отношение, которые можно записать как A ⇒ B, то некоторые объекты класса A будут неявным образом унаследованы и могут быть использованы в классе B. Если некоторый объект X объявлен в классе A и не переопределен в классе B, то любая ссылка на объект X внутри класса B в силу наследования фактически является ссылкой на объект X из класса A, что аналогично ссылке на переменную i в приведенном выше примере из языка C.

Если между классами A и B определено отношение A ⇒ B, то говорят, что класс A является *родительским* классом, или *суперклассом*, а класс B — *зависимым*, или *дочерним*, классом, или *подклассом*. Класс A является прямым *предком*, или *родителем* класса B. На рис. 7.2, *а* класс A является прямым предком для классов B и C, которые друг для друга являются ближайшими родственниками одного «поколения». Классы B и C являются прямыми потомками класса A, тогда как класс D является просто потомком класса A. Если некоторый класс имеет только одного родителя, то такое наследование называется *простым*. Если же класс имеет нескольких родителей, то такая форма наследования называется *множественной*.

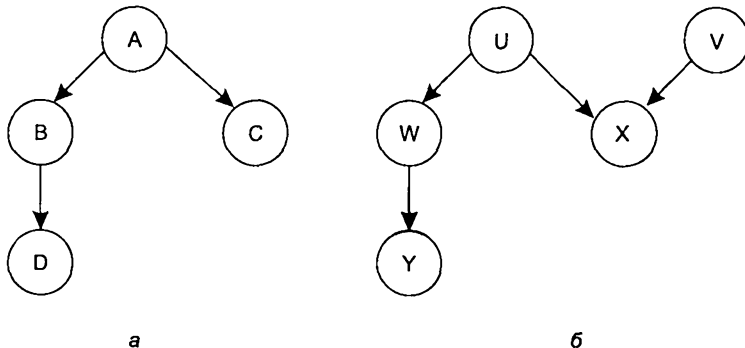


Рис. 7.2. Наследование: а — простое; б — множественное

Множественное наследование допустимо в C++, но не в Java. В языке Ada 95 добавлены помеченные (tagged) типы, реализующие форму наследования, при которой объекты, объявленные как tagged private, могут быть использованы в новом пакете с дополнительными компонентами. Ниже приводится несколько примеров:

```
type Calendar is tagged private:
...
private
type Calendar is tagged record
      Day: integer:
      Month: integer:
      end record
```

и

```
type AnnualCalendar is new Calendar with record
      Year: integer:
      end record
```

7.2.1. Производные классы

Классы в языках, подобных C++, Java и Smalltalk, тесно связаны с концепцией инкапсуляции. Обычно класс состоит из двух частей: одна может быть унаследована каким-либо другим классом, а другая остается скрытой от внешних воздействий и используется только внутри этого класса. Например, стеки целых чисел в Ada (см. листинг 7.4) могут быть представлены в C++ как

```
class intstack
      {private:
      int size:
      int storage(100): }
```

или в Java

```
class intstack
      {private int size:
      int [] storage = new int[100]: }
```

Имя intstack — это имя класса, известное за его пределами, в то время как объявления переменных size и storage, следующие за модификатором переменных private, как и в языке Ada, представляют собой компоненты класса intstack, которые известны только внутри класса.

В понятие абстракции типа данных, как говорилось выше, входят и описания данных, и функции, которые могут оперировать данными этого типа. Такие функции часто называются *методами*. Более полное описание класса intstack, в который включены методы, могло бы выглядеть следующим образом:

```
class intstack
      {public:
      intstack() {size=0:}
      void push(int i) { storage[size++]=i:}
      int pop …
   private:
      int size:
      int storage(100):
      }
```

Модификатор public означает, что соответствующие имена видимы за пределами определения класса и могут быть унаследованы другими классами. Входящая в определение класса функция intstack(), имя которой совпадает с именем класса, является специальной функцией, называемой *конструктором*, и вызывается каждый раз, когда создается объект этого класса. В данном случае эта функция инициализирует пустой стек, но, вообще говоря, она может выполнить любой код инициализации, необходимый для создания объектов этого класса. В этом примере функции push и pop могут быть вызваны и использованы за пределами определения класса как методы объектов класса intstack, например b.push(7) или j = b.pop() для стека b. Имена size и storage, напротив, известны только в пределах определения класса.

Для уничтожения объекта класса intstack и высвобождения занимаемого им места в памяти следовало бы определить специальную функцию ~intstack, которая называется *деструктором*. В приведенном примере деструктор не нужен. Но если бы конструктор отводил в памяти какое-то дополнительное место, требуемое для объектов класса, то деструктор был бы необходим для его освобождения впоследствии.

Запись b.push(7) не так уж необычна, как может показаться на первый взгляд. Определения классов похожи на определения типов в языке C, за тем исключением, что в определение класса входят еще и компоненты-функции, манипулирующие объектами этого класса. Таким образом, b.push — это компонент-функция push объекта b. Сходство между понятием класса в C++ и Java и конструкцией sruct в языке C не является случайным. Объявление в языке C

srtuct A {int B} —

это просто сокращенная запись объявления C++

class A {public: int B}

В определение класса, таким образом, могут входить и определения функций, и определения данных.

Наследование в Java и C++ происходит при помощи *производных* классов. Например, один класс может содержать объекты типа elem, а в другом классе эти объекты могут использоваться для создания стеков, как показано в листинге 7.6. Имя ElemStack, которое следует за ключевым словом class, представляет производный класс, а имя, непосредственно следующее за двоеточием после имени класса ElemStack, обозначает базовый класс (имя elem). (В языке Java для указания базового класса вместо двоеточия используется ключевое слово extends.) Если рассматривать операции класса просто как компоненты определения объектов этого класса, то понять синтаксис достаточно легко. Запись x.ToElem(...) становится ссылкой на метод, определенный в классе elem, в результате которой вызывается процедура ToElem, неявным аргументом которой является объект x.

Листинг 7.6. Производные классы в C++

```
class elem {
public:
      elem() { v=0:}
      void ToElem(int b) { v = b:}
      int FromElem() { return v: }
private:
      int v: }

class ElemStack: elem {
public:
      ElemStack() { size=0:}
      void push(elem i)
         { size=size+1: storage[size]=i:}
      elem pop()
         { size=size-1: return storage[size+1]}
private:
      int size:
      elem storage[100]: }
   { elem x:
      ElemStack y:
```

```
        int i;
        read(i);       - Получаем целочисленное значение для i
        x.ToElem(i);   - Преобразуем x к типу elem
        y.push(x);     - Помещаем x в стек y
        ...
}
```

Этот пример иллюстрирует некоторые важные аспекты использования классов для инкапсуляции данных:

1. Все открытые (public) имена в классе elem наследуются в классе ElemStack. Эти имена также являются открытыми в ElemStack и известны для любого пользователя этого стека.
2. Содержимое объекта типа elem является закрытой (private) информацией и не известно за пределами определения класса. В нашем примере ToElem и FromElem действуют как операции приведения типов, которые преобразуют объекты целочисленного типа к типу elem. Такие операции нужны всегда, если внутренняя структура класса закрыта.

Ключевые слова private и public используются для определения области видимости наследуемых объектов. В языке Java они являются атрибутами объектов данных. В C++ и Java имеется также ключевое слово protected, означающее, что имя с этим модификатором является видимым для любых типов, производных от базового типа, но не известно вне иерархии определения классов.

Реализация. Реализация классов не прибавляет слишком много работы транслятору. Для производного класса к пространству локальных переменных добавляются наследуемые имена базового класса, и только открытые имена делаются видимыми для пользователей этого класса. Истинное распределение памяти для определенных объектов может быть осуществлено статически с использованием объявлений данных в определении класса.

Если в определении класса присутствует конструктор, то транслятор должен включать в создаваемый код обращение к этой функции каждый раз, когда он встречает объявление объекта этого класса (например, при входе в блок). Наконец, вызов методов, например x.push(i), обрабатывается транслятором как вызов функции push(x, i), где x рассматривается как неявный первый аргумент этой функции. Управление памятью такое же, как в стандартном C; в C++ также может быть использована та же самая, что и в C, основанная на стеках организация.

Реализовать управление памятью в языке Java несколько сложнее, так как массивы в Java являются динамическими. Объявление массива и его размещение в памяти осуществляются при помощи оператора new. Например,

```
int[] NewArray = new int[10];
```

создает массив с именем NewArray, а операция new динамически выделяет пространство памяти для десяти целых чисел, которые составят элементы этого массива. Таким образом, NewArray является просто указателем на это место в памяти. Виртуальная машина Java для динамического размещения массивов использует специальную область памяти, называемую кучей.

Для каждого экземпляра класса выделяется собственная область памяти, состоящая из объектов данных, формирующих объекты класса, а также указателей на все методы, определенные для объектов этого класса. Если некоторый объект

является производным от некоторого базового класса, то транслятор копирует все детали реализации этого класса в то место памяти, которое отведено для фактического размещения создаваемого объекта. Этот подход к наследованию называется *основанным на копировании* и является наиболее простой и очевидной моделью его реализации.

Альтернативным вариантом реализации является *основанный на делегировании* подход. В этом варианте любой объект производного класса будет использовать область памяти данных базового класса. Наследуемые свойства базового класса не дублируются в производном объекте. Эта модель подразумевает некую форму совместного использования данных, которая позволяет автоматически изменять производный объект в соответствии с изменениями базового объекта.

В C++ наследование реализовано с использованием основанного на копировании подхода, но надо отметить, что альтернативный способ более эффективно использует ресурсы памяти (благодаря совместному использованию наследуемых свойств объектов разными программами) и позволяет автоматически и мгновенно распространять изменения одного объекта по всей иерархии производных классов.

Множественное наследование

В этом разделе при обсуждении классов в языке C++ предполагалось, что иерархия классов имеет древовидную структуру со множеством производных классов, наследующих данные и функции, определенные в родительских классах. Это модель, реализованная в языке Smalltalk, который стал первым объектно-ориентированным языком (см. раздел 7.2.5). Но в языке C++ допускается построение новых классов, производных одновременно от нескольких родительских классов. В объявлении

```
class A:B,C {…}
```

класс A является производным как от класса B, так и от класса C. До тех пор пока множества объектов, определяемых классами B и C, не перекрываются, никаких проблем с объединением их в одном классе A не возникает. Также не возникает никаких дополнительных сложностей при реализации классов любым из описанных выше способов.

7.2.2. Методы

Термин *объектно-ориентированное программирование* в последние годы был предметом такой шумной рекламы, что быстро утратил свое истинное значение. Для многих людей этот термин стал синонимом концепции инкапсуляции, которая обсуждалась в предыдущих разделах и главах. Но объектная ориентация означает все же нечто большее, чем просто объединение данных и подпрограмм в единый модуль. Наследование методов при создании новых объектов дает дополнительные возможности, выходящие за пределы простой инкапсуляции.

Рассмотрим, например, как можно расширить наш пример с классом `ElemStack`, приведенный выше. В определение класса, представленное в листинге 7.7, мы добавили открытую процедуру `MyType`, которая выводит на печать имя типа: `I am type ElemStack`; а также сделали внутреннюю структуру класса `ElemStack` *защищен-*

ной (protected), чтобы она могла наследоваться любым производным от нашего классом.

В наш класс стека входили только операции push и pop, которые соответственно добавляли и удаляли элементы стека. Предположим, что нам нужен новый класс NewStack, который функционирует так же, как класс ElemStack, но включает еще один метод peek, который возвращает значение верхнего элемента стека, никак не изменяя при этом сам стек. При помощи определения класса NewStack из листинга 7.7 эта цель, по-видимому, достигается. Действительно, класс NewStack наследует все свойства класса ElemStack (все открытые методы и данные), но в него добавлен новый метод peek. Хотя в определении класса NewStack ничего не сказано об операциях push и pop класса ElemStack и операциях ToElem и FromElem класса elem, эти операции автоматически становятся применимыми к объектам класса NewStack. Если эти классы поддерживаются как отдельные модули, разрабатываемые разными программистами, изменения в спецификации (а следовательно, и в реализации) или класса elem, или класса ElemStack будут автоматически перенесены и в определение класса NewStack и будут оказывать такое же действие на объекты класса NewStack, как и на объекты класса ElemStack.

Листинг 7.7. Наследование методов

```
class elem {
public:
      elem() { v=0;}
      void ToElem(int b) { v = b;}
      int FromElem() { return v; }
private:
      int v; }

class ElemStack: elem {
public:
      ElemStack() { size=0;}
      void push(elem i)
      { size=size+1; storage[size]=i;}
      elem pop()
      { size=size-1; return storage[size+1]}
      void MyType() {printf("I am type ElemStack \ n")}
protected:
      int size;
      elem storage[100]; }
class NewStack: ElemStack {
public:
      int peek() {return storage[size].FromElem() }}
```

Тем не менее некоторая проблема остается нерешенной. Дело в том, что метод MyStack для объектов типа NewStack по-прежнему печатает I am type ElemStack, поскольку этот метод унаследован от класса ElemStack. Эту проблему можно решить одним из следующих двух способов.

1. Можно просто переопределить метод MyType в классе NewStack следующим образом:

   ```
   void MyType() {print("I am type NewStack\n")}
   ```

   Хотя этот способ и работает, он все же не является оптимальным, поскольку требует в производном классе полного переопределения каждого метода, в котором необходимы изменения.

2. Можно использовать виртуальную (virtual) функцию. При определении метода обычно каждое имя подпрограммы, которая вызывается в определении, связывается с подпрограммой, на которую оно ссылается, *во время определения метода*. Это стандартная методика синтаксического связывания, присутствующая в языках типа C, Pascal, Ada и FORTRAN и большинстве компилируемых языков. Но *виртуальные* подпрограммы динамически *связываются во время вызова подпрограммы.*

Чтобы понять разницу, определим метод MyType внутри класса ElemStack следующим образом:

```
virtual void TypeName() {printf("ElemStack\n"):}:
void MyType() {printf("I am type"). TypeName():}
```

В классе NewStack можно определить TypeName следующим образом:

```
virtual void TypeName() {printf("NewStack\n"):}:
```

Хотя в приведенном примере разница между двумя подходами не очень ярко выражена, отсроченное, позднее связывание вызова виртуального метода позволяет динамически изменять поведение классов во время выполнения. Такой подход был бы более важен в классах, где методы, подобные MyType, исключительно длинные и сложные. Вместо того чтобы полностью дублировать этот метод с необходимыми модификациями в каждом производном классе, требуется переопределить всего лишь небольшую виртуальную функцию, используемую в методе и отражающую необходимые изменения.

Реализация. Виртуальные методы можно реализовать в некотором смысле аналогично центральной таблице окружения записей активаций (раздел 9.4.1). Для каждого виртуального метода в производном классе резервируется некоторая область внутри записи, определяющей этот класс. Процедура-конструктор просто заполняет эту область, помещая туда ссылку на новую виртуальную процедуру, если она определена для данного класса. В противном случае в эту область помещается ссылка на виртуальную процедуру из базового класса.

7.2.3. Абстрактные классы

Может возникнуть ситуация, когда нам нужно, чтобы наше определение класса было всего лишь простым шаблоном для класса и с его помощью не было бы возможно объявлять объекты. Для реализации подобной модели существует два альтернативных способа: абстрактные суперклассы и *смешанное* (mixin) наследование.

Абстрактный суперкласс. Рассмотрим класс ElemStack с виртуальной функцией TypeName, определенный выше. Если он определен, то пользователь имеет право включить в программу оператор

```
ElemStack X:
```

и создать экземпляр класса ElemStack. Однако нам может понадобиться, чтобы класс ElemStack был просто шаблоном суперкласса и чтобы все объекты, использующие этот класс, принадлежали к производным от него классам. В C++ это можно сделать, объявив TypeName как чисто виртуальную функцию:

```
virtual void TypeName()=0:
```

Невозможно создать объект класса, содержащего хотя бы одну чисто виртуальную функцию. Любой производный класс должен переопределить эту функцию для того, чтобы можно было создавать экземпляры этого класса.

Смешанное наследование. До сих пор предполагалось, что наследование происходит по принципу A ⇒ B, так что класс B является производным и в то же время модификацией класса A. Имеется и другая форма наследования, называемая *смешанной* (mixin), при использовании которой мы только определяем различие между базовым классом и новым производным классом. Вернемся к нашему примеру и снова рассмотрим базовый класс ElemStack и производный от него класс NewStack. Вместо того чтобы определять новый класс для класса NewStack, можно определить *дельта-класс*, содержащий отличия производного класса от базового. Хотя на самом деле в языке C++ такая возможность не предусмотрена, мы воспользуемся обозначениями, принятыми в этом языке, для определения дельта-класса StackMode:

```
deltaclass StackMod
      {int peek() {return storage[size].FromElem():}
      }
```

Затем мы создали бы новый класс NewStack следующим образом:

```
class NewStack = class ElemStack + deltaclass StackMod
```

Смысл этой записи в том, что новый класс наследует все свойства класса ElemStack в том виде, как они были модифицированы дельта-классом StackMod.

Преимущество смешанного наследования заключается в том, что дельта-классы могут применяться с любым классом. Так, если бы у нас имелся некоторый сравнимый класс ElemQueue, то мы могли бы, используя тот же дельта-класс, что и в предыдущем примере со стеками, создать новый класс, который позволял бы получить значение элемента в конце очереди:

```
class newqueue = class ElemQueue + deltaclass StackMod
```

Этот дельта-класс можно было бы применять в различных ситуациях, избегая таким образом необходимости сложного переопределения всех объектов класса.

7.2.4. Обзор языка Smalltalk

История. Язык Smalltalk отличается от всех других языков, описанных в этой книге, в двух отношениях:

1. Он разрабатывался как целая система, а не как просто средство для написания программ.
2. Объектная ориентация в этом языке изначально была встроенной концепцией в противоположность языкам C++ и Ada, в которых к уже существующим механизмам была просто добавлена концепция наследования.

Использование языка Smalltalk исходно было ограничено отсутствием доступных реализаций. Но теперь это уже не так. Язык Smalltalk имеет немногочисленных, но преданных приверженцев.

Smalltalk был разработан в начале 70-х гг. Аланом Кеем (Alan Kay) в исследовательском центре Xerox PARC (Palo Alto Research Center). Целью проекта под названием Dynabook было создание целой среды для персонального компьютера.

Это происходило во время расцвета деятельности Xerox PARC — тогда были разработаны персональные компьютеры Alto и Star, мышь, оконная среда (впоследствии весьма эффективно использованная Стивеном Джобсом (Stiven Jobs) в фирме Apple для разработки компьютеров Macintosh), значки для программ, сети Ethernet и Smalltalk. В 1972 г. Дан Ингаллс (Dan Ingalls) разработал первую реализацию языка, известную как Smalltalk-72. Это была первая практическая версия языка. Затем в Smalltalk были внесены некоторые изменения и общепринятым стандартом описания языка стал Smalltalk-80. Поскольку официального стандарта не существует, название *Smalltalk* применяется для обозначения набора нескольких достаточно близких и чрезвычайно совместимых языков.

Первоначально для выполнения программ на Smalltalk требовалась целая вычислительная среда компьютера специального назначения. По этой причине распространение этого языка было достаточно ограниченным. Тем не менее в настоящее время существует несколько реализаций этой системы общего назначения, и ее можно легко устанавливать на машины широкого класса типа рабочих станций UNIX и PC. В нашей книге мы описываем язык (а также реализацию, использованную для тестирования представленных в книге программ), соответствующий версии GNU Smalltalk 1.1.1, разработанной Фондом свободно распространяемого программного обеспечения (Free Software Foundation), которая легко доступна из многочисленных источников.

Краткий обзор языка. Уникальность языка Smalltalk определяют следующие свойства, благодаря которым он отличается от остальных языков, описанных в этой книге.

- *Среда разработки.* Smalltalk разрабатывался как целая система: язык, компьютер и среда программирования. Однако в данной книге мы рассматриваем Smalltalk только как язык программирования и совершенно не касаемся вопросов, связанных со средой программирования.
- *Минимальный язык разработки.* Небольшой базовый язык определяет Smalltalk. По существу, все, что можно делать в Smalltalk, — это разработать класс как подкласс других классов и к каждому из разработанных классов присоединить методы. Основные операторы Smalltalk определяют присваивание и последовательное выполнение действий. Истинная мощь языка заключается в механизме параметрических вызовов, называемых *сообщениями.* При помощи сообщений можно определять структуры управления, подобные нашим обычным конструкциям `if` и `while`.

  Smalltalk поставляется с предопределенным набором определений классов, написанных на языке Smalltalk. Когда Smalltalk устанавливается на компьютер, то создается образ пустого окружения путем включения всех этих определений. При вызове интерпретатора Smalltalk вы получаете свою собственную копию этого образа. Внутри этого образа вы строите новые классы и модифицируете существующие классы добавлением или изменением методов. Системному администратору легко построить систему Smalltalk с альтернативным набором предопределенных определений классов, так что не совсем понятно, что следует понимать под языком Smalltalk. В данной книге мы называем языком Smalltalk набор предопределенных классов.

- *Выполнение программы Smalltalk.* Модель выполнения для Smalltalk основана на коммуникационной модели (communication model). Данные в Smalltalk состоят из объектов, а методы рассматриваются как сообщения, посылаемые объектам. Так, 1 + 2 означает, что сообщение «+» с параметром 2 послано целочисленному объекту 1. Метод «+» в данном случае возвращает значение 3. На первый взгляд это может показаться странным, но становится вполне естественным после небольшой практики.

  Smalltalk использует динамическую модель последовательности выполнения действий. Каждый метод выполняется с одновременным созданием записи активации, называемой *контекстом.* Так как блоки, которые содержат локальные переменные, могут быть присвоены переменной, то стековый механизм организации памяти, при котором читается последний записанный объект, становится здесь некорректным. Как для хранения объектов данных, так и для хранения записей активации Smalltalk должен использовать динамическую организацию памяти в виде кучи. Как правило, используется динамическая сборка мусора, хотя семантика необходимой сборки мусора в языке не специфицирована.

7.2.5. Объекты и сообщения

Smalltalk представляет альтернативный подход к разработке объектов и методов, значительно отличающийся по своей идеологии от моделей наследования, представленных ранее в этой главе для языков Ada и C++. Идея Smalltalk принадлежит Алану Кею из Xerox Palo Alto Research Center, он предложил ее в начале 70-х гг., но окончательный вариант вобрал в себя многочисленные дополнения других разработчиков. Smalltalk был разработан как целая вычислительная среда для персонального компьютера. Поэтому он включал в себя не только язык для представления алгоритмов, но и среду вычисления, состоящую из разделенного на окна (windows) экрана монитора и мыши рабочей станции Xerox Alto. Теперь подобные возможности являются общепринятыми, но в то время их появление рассматривалось как революционный шаг вперед.

Программа на Smalltalk состоит из набора определений классов, состоящих, в свою очередь, из объектов данных и методов. Все данные инкапсулированы, так как только методы, определенные для класса, имеют доступ к данным, принадлежащим этому классу. Сокрытие информации и инкапсуляция являются неотъемлемыми встроенными свойствами языка в отличие от C++, в котором эти свойства были добавлены поверх уже существующей структуры типов языка.

Программа на Smalltalk обычно включает в себя три основные составляющие.

1. *Определения классов.* Они представлены выполняемыми операторами, определяющими внутреннюю структуру и методы, которые могут быть использованы для создания и манипуляции объектами класса. Также могут быть определены данные, используемые совместно всеми объектами определенного класса.
2. *Создание объектов.* Для каждого определения класса создаются свои объекты посредством вызова методов создания в определении этого класса. Методы можно определять для отдельных экземпляров класса.

3. *Передача сообщений.* Для выполнения действий объектам передаются методы в форме сообщений. Вместо того чтобы связывать с функцией набор параметров, как это делается в большинстве других языков, в Smalltalk функция (то есть метод) связывается с объектом данных. Эта связь метода с объектом называется *сообщением.*

В Smalltalk существуют три типа сообщений:

1. *Унарное* сообщение — это метод, не имеющий параметров. Например, для создания объектов в большинстве классов используется встроенный метод new:

   ```
   x _ Set new
   ```

   В данном случае переменной x присваивается вновь создаваемый объект класса Set. Теперь x является экземпляром объекта класса Set.

2. *Бинарное* сообщение, как правило, используется для арифметических операций. Выражение 3 + 6 соответствует бинарному методу «+», посланному объекту 3 с параметром 6. Метод «+» возвращает объект 9 как результат данного сообщения.

3. *Ключевые* сообщения ведут себя примерно так же, как перегруженные функции в языках типа Ada и C++. Например, для присвоения значения третьему элементу массива x следует сначала создать массив с нужным количеством элементов, а затем выполнить операцию присваивания:

   ```
   x _ Array new:10
   x at:3 put:42
   ```

   Сначала метод new: с параметром 10 посылается классу Array для создания массива из десяти элементов, который присваивается переменной x. Вызывается ключевой метод at: и put: для присваивания третьему компоненту массива x значения 42. Название at:put: метода этого оператора присваивания образуется путем конкатенации ключевых слов (at: и put:). Другой метод, at:, посланный экземпляру массива, извлекает значение соответствующего элемента массива.

Использование символа «_» для обозначения операции присваивания объясняется некоторыми фактами из истории развития компьютерной техники. Исходно операция присваивания в Smalltalk обозначалась символом ←, который в 60-е и 70-е гг. был представлен на клавиатурах той же клавишей, что и символ «_», поэтому внутреннее определение этих символов было одинаковым. Со временем символ ← исчез с клавиатуры, но в Smalltalk используется символ, который соответствует той же клавише. Аналогично операция, указывающая возвращаемое каким-либо методом значение, исходно обозначалась символом ↑, который размещался на той же клавише, что и символ «^». Со временем этот символ также исчез с клавиатуры.

Последовательности операторов в Smalltalk могут быть выполнены как блоки, например:

```
[ :локальная_переменная | оператор1 ... операторn]
```

где локальная_переменная — это необязательная локальная переменная, объявленная в блоке. (Первый символ : в блоке необходим для устранения синтаксической неоднозначности в определении блоков.) Выполнение блока операторов происхо-

дит посредством передачи метода value этому блоку, а результатом выполнения является последнее выражение блока. Так

```
| x |
x _ ['Это строка'].  "Переменной x присвоен данный блок."
x value print !      "Вычисляется блок переменной x."
```

означает следующее:

1) объявляется локальная переменная x;

2) переменной x присваивается блок;

3) метод x value возвращает строку Это строка, которая печатается методом print.

В Smalltalk комментарии обозначаются двойными кавычками. Символ ! в Smalltalk обозначает команду для выполнения предыдущей последовательности операторов.

Использование передачи ключевого параметра позволяет создавать многие широко используемые структуры управления. Предопределенное окружение Smalltalk включает встроенный метод для работы с булевыми данными ifTrue:ifFalse:. Каждое из этих ключевых слов получает в качестве параметра некоторый блок данных. Например, true ifTrue:ifFalse: вычислит блок ifTrue:, тогда как false ifTrue:ifFalse: вычислит блок ifFalse:. Записывая эти конструкции с использованием общепринятых отступов, подобных тем, что применяются в языке Pascal, мы получим следующий синтаксис:

```
x > 2
      ifTrue: ['x больше 2' printNl]
      ifFalse:['x меньше или равен 2' printNl]
```

В данном случае метод > с параметром 2 передается объекту x. Метод > возвращает объект true или false в зависимости от значения x. Этому булевому объекту затем передается ключевой метод ifTrue:ifFalse: с двумя блоками в качестве параметров. В зависимости от того, какой был возвращен булев объект, выполняется блок ifTrue: или ifFalse:. (Метод printNl аналогичен методу print, за исключением того, что после напечатанного объекта он печатает также символ новой строки.) Алгоритм вычисления очень похож на вычисление оператора if-then-else, но фактическое выполнение радикально отличается. Используя этот подход, можно аналогичным образом разработать циклические конструкции.

Smalltalk — единственный из описанных в нашей книге языков, полностью основанный на концепциях инкапсуляции и абстракции. Наследование в Smalltalk является базовой характеристикой процесса вызова методов.

Неупорядоченность в определении методов приводит к другой проблеме в определении методов — неоднозначности. Например, рассмотрим следующие методы:

```
!Datastore class methodsFor: 'Misc'!
asgn: aValue to: bValue
      aValue printNl !
to: aValue asgn: bValue
      aValue printNl !!
```

На первый взгляд это выглядит как неоднозначное определение метода asgn:to:. Тем не менее to:asgn: — это совсем другой метод:

```
st> Datastore asgn: 8 to: 9 !
Execution begins...
```

```
8
st> Datastore to: 9 asgn: 8 !
Execution begins...
9
```

Очень важно, чтобы каждый ключевой метод использовал уникальный набор имен.

У языка Smalltalk, как было сказано, имеется некоторый (хотя и небольшой) круг сторонников. Этот язык достаточно интересен, но лишь немногие коммерческие системы были написаны на этом языке. Недостаточное количество доступных трансляторов Smalltalk до сих пор затрудняло его распространение. Возможно, теперь, когда трансляторы стали легко доступны, ситуация изменится.

Наследование классов

Данные в Smalltalk организованы на основе иерархии классов. Если какой-либо метод, который передается объекту, не определен внутри этого класса, то он передается родительскому классу и т. д. Класс Object является родительским суперклассом для всех классов. Наследование методов соответственно является основным свойством языка. Однако допускается только простое наследование с одним родительским классом для каждого дочернего класса, хотя методы могут передаваться через несколько уровней родительских классов.

Для ключевых методов параметры явным образом указаны в объявлении метода, как, например, в следующем случае:

```
ifTrue: trueBlock ifFalse: falseBlock
   "Переменные trueBlock и falseBlock являются параметрами
   в теле метода."
```

Но каким образом можно получить доступ к объекту, которому передан метод? Например, если x > 2, каким образом метод > получает доступ к объекту x и получает его значение? Для этого существует объект self. Он ведет себя примерно так же, как параметр this в C++.

Необходимость объекта self и его использование в иерархии объектов можно продемонстрировать на следующем простом примере. Предположим, нам требуются два класса: ClassA и ClassB, являющийся подклассом ClassA. Эти классы можно определить следующим образом:

```
Object subclass: #ClassA
   instanceVariableNames: ' '
   classVariableNames: ' '
   poolDictionaries: ' '
   category: nil !
ClassA subclass: #ClassB
   instanceVariableNames: ' '
   classVariableNames: ' '
   poolDictionaries: ' '
   category: nil !
```

ClassA определен как подкласс класса Object (просто для того, чтобы поместить его в какое-то место в иерархии Smalltalk), а ClassB является подклассом ClassA. instanceVariableNames определяет набор локальных имен, используемых в каждом отдельном экземпляре объекта этого класса, в то время как classVariableNames — это глобальные данные, которые используются во всех экземплярах объектов это-

го класса. Конструкции poolDictionaries и category в данном случае не нужны (они описаны в приложении, раздел П.12), но синтаксис Smalltalk требует, чтобы все параметры ключевых методов были приведены, даже если это пустые параметры.

Рассмотрим методы printIt и testIt, которые мы добавляем к классу ClassA при помощи команды methodsFor:

```
!ClassA methodsFor: 'basic'!
      "Добавляем методы к классу ClassA"
printIt
      'Это класс A' printNl !
testIt
      self printIt !!
```

Каждый метод задается посредством указания его имени (и, если требуется, параметров), затем следует его определение. Окончание определения фиксируется символом !. Символ !! обозначает завершение объявлений methodsFor для ClassA.

Применение этих методов можно проиллюстрировать на следующем примере:

```
| x | x _ ClassA new. x testIt !
```

что, естественно, выводит на печать следующее сообщение:

```
Это класс A
```

При этом выполняются следующие шаги:

1) метод new передается классу ClassA, который, в свою очередь, передает его родительскому классу Object;
2) Object создает новый объект и присваивает его переменной x;
3) метод testIt передается созданному объекту x класса ClassA;
4) метод testIt определяется как self printIt, где self относится к объекту x;
5) метод printIt передается объекту x, который и печатает требуемое сообщение.

Рассмотрим теперь, что произойдет, если мы определим аналогичный метод для ClassB, в котором есть метод printIt, но отсутствует testIt:

```
!ClassB methodsFor: 'basic'!
printIt
      'Это класс B' printNl !!
```

Если мы напишем:

```
| x | x ClassB new. x testIt !
```

мы получаем:

```
Это класс B
```

Метод testIt передается экземпляру x класса ClassB. Поскольку в классе ClassB метод testIt не определен, то этот метод передается родительскому классу ClassA. self по-прежнему указывает на объект x, но он теперь принадлежит классу ClassB, поэтому вызывается метод printIt класса ClassB, а не родительский метод testIt класса ClassA.

7.2.6. Концепции абстракций

После обсуждения роли методов и виртуальных функций, вероятно, полезно вернуться к обсуждению роли абстракций в языках программирования. Инкапсуля-

ция часто рассматривается как механизм, позволяющий осуществлять контроль в процессе разработки программы по принципу «разделяй и властвуй». Программист имеет доступ только к тем объектам данных, которые являются частью спецификации именно того сегмента программы, разработкой которого он занимается. Спецификация других частей программы (и внутренняя реализация данных, отвечающих этой спецификации) от него скрыта. Тем не менее абстракцию данных и связанную с ней концепцию наследования не следует рассматривать только как информационную стену, преграждающую программисту доступ к содержимому недоступных объектов данных.

Наследование предоставляет механизм обмена информацией между объектами в связанных классах. Если A ⇒ B означает, что B — это класс, связанный с классом A, то каково взаимоотношение между объектами этих классов? Возможны четыре типа взаимоотношений, как показано на рис. 7.3. К этим четырем типам сводятся различные способы использования наследования в языках программирования.

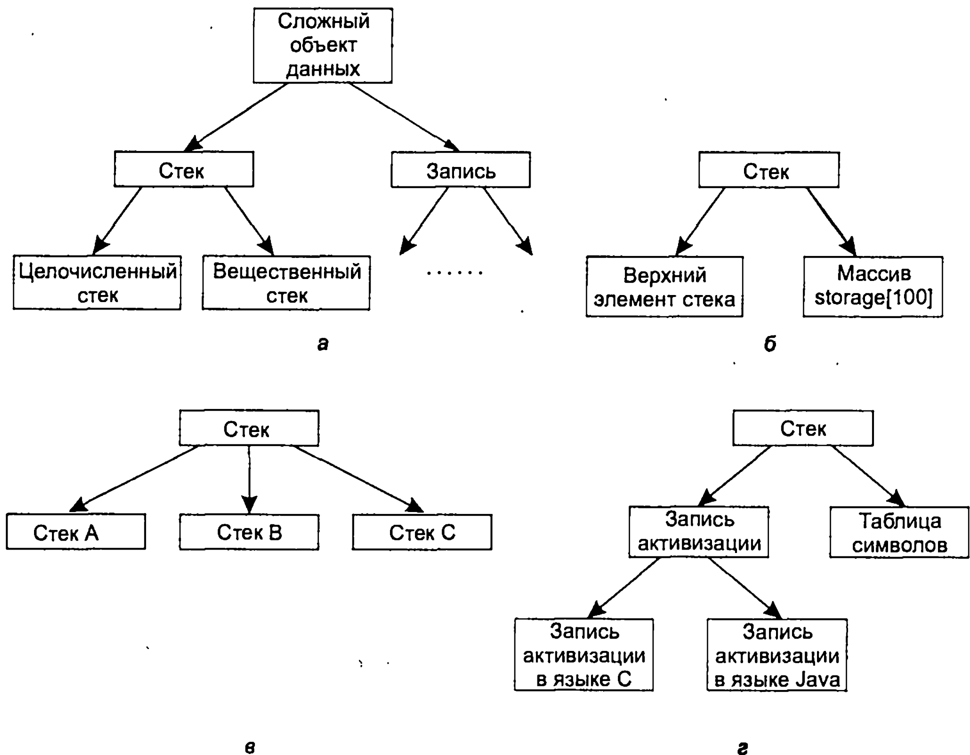


Рис. 7.3. Концепции абстракций: *а* — конкретизация; *б* — декомпозиция; *в* — создание; *г* — индивидуализация

Конкретизация. Это наиболее распространенная форма наследования, которая позволяет уточнять свойства производного объекта B по сравнению с исходным объектом A (см. рис. 7.3, *а*). В частности, *стек* (stack) — это более точная характеристика объекта, чем *сложный* объект данных, а указание на тип содержимого

стека (int stack, целые числа) уточняет, какой именно стек мы рассматриваем. В то же время все основные свойства стека сохраняются и в случае целочисленного стека, так как наследуются методы push и pop. Ранее мы описывали класс NewStack как производный от ElemStack с добавлением функции peek. Можно сказать, что класс NewStack является конкретизацией класса ElemStack.

Противоположной концепции конкретизации является концепция обобщение (например, *стек* (stack) — это более общее понятие, чем int stack — целочисленный стек, или real stack — вещественный стек). Обобщение представляет суперкласс для набора производных классов.

Декомпозиция. Это принцип разбиения абстракции на компоненты (см. рис. 7.3, *б*). Он является типичным механизмом инкапсуляции в языках типа Ada, в которых отсутствует наследование методов. Например, стек (внешняя абстракция) состоит из переменной top (верхний элемент стека) и массива storage[100] (внутренняя абстракция). Внутренние имена неизвестны за пределами определяющего класса. Противоположная концепция называется *агрегацией*.

Создание экземпляров. Этот принцип заключается в создании экземпляров класса. По существу, он сводится к операции *копирования* (см. рис. 7.3, *в*). Объявление экземпляров объекта некоторого класса в программе на C++ является типичным примером использования принципа *создания экземпляров* и заключается в том, что в программе объект x объявляется принадлежащим к типу y:

```
stack A, B, C;
```

В данном случае среда выполнения создает три копии объекта типа stack и называет их A, B и C.

Противоположная концепция называется *классификацией*. В частности, в нашем примере мы могли бы классифицировать A, B и C как экземпляры класса stack.

Индивидуализация. Этот четвертый принцип абстракции (см. рис. 7.3, *г*) несколько труднее для понимания, чем три предыдущих. В данном случае сходные объекты группируются вместе для некоторых общих целей. Противоположная концепция называется *группированием*. Например, среди стеков, используемых транслятором, могут присутствовать записи активации и таблицы символов. И в С, и в Pascal записи активации реализованы как стеки, но эти стеки имеют различные характеристики и являются примерами *конкретизации*. Все стеки тем не менее имеют одни и те же основные характеристики, примерами которых могут быть методы push и pop. Но, несмотря на сходство используемых структур, роли записей активации при выполнении программы на языках С и Pascal различны.

7.3. Полиморфизм

Использование параметров в подпрограммах — одна из наиболее ранних характеристик языков программирования. Тем не менее в большинстве языков программирования параметры имеют одну важную характеристику: у них есть *l*-значение. Это значит, что параметр представляет собой объект данных, который должен быть размещен в записи активации во время выполнения программы. Всем знакомы конструкции типа P(A, 7) или Q(B, false, 'i'). Но выражения типа

R(integer, real), в которых параметрами являются типы данных, встречаются крайне редко.

Полиморфизм означает способность одной операции или имени подпрограммы ссылаться на любое число определений функций, зависящих от типа данных параметров и результатов. Мы уже сталкивались с некоторым ограниченным видом полиморфизма в главе 5 при обсуждении перегрузки некоторых операций. В выражениях 1 + 2 и 1.5 + 2.7 имеются в виду две различные операции сложения: в первом случае это целочисленное сложение, во втором — вещественное. Использование модификатора generic в языке Ada позволяет с некоторыми ограничениями перегружать функции, когда компилируется несколько экземпляров функции, каждому из которых соответствует свой набор типов параметров и результатов. Операция унификации в языке Prolog также может рассматриваться как вид полиморфизма, так как Prolog пытается унифицировать запрос при помощи набора правил и фактов, содержащихся в его базе данных.

Полиморфизм, как правило, применяется к функциям, у которых одним из параметров является тип данных. Из обсуждаемых языков наиболее общее применение полиморфизма обеспечивают ML и Smalltalk. Например, аргумент тождественной функции ident может быть любого типа. Если мы определим ее как

```
fun ident(x) = x;
```

мы получим от ML ответ:

```
- fun ident(x) = x;
val ident = fn : 'a -> 'a
```

означающий, что функция ident получает аргумент типа 'a (то есть произвольного типа) и возвращает аргумент типа 'a (то есть того же типа) для параметра типа 'a. Таким образом, передача в функцию ident целочисленных и строковых параметров, а также целочисленных списков является корректной:

```
- ident(3);
val it = 3 : int
- ident("abc");
val it = "abc" : string
- ident([1,2,3]);
val it = [1,2,3] : int list
```

В ML некоторые операции в определении функции допускают полиморфизм, как, например, операция равенства и операции, создающие списки объектов. Тем не менее для некоторых операций область возможных типов параметров ограничена. Например, арифметические операции сложения и вычитания могут применяться только к арифметическим типам. Как уже было объяснено ранее, операция сложения + перегружена, так как она относится к различным числовым типам данных. По этой причине уже рассмотренные нами выше определения в ML типа

```
fun area(length,width) = length*width;
```

являются неоднозначными, хотя area(1, 2) или area(1.3, 2.3) будут уже вполне однозначными обращениями к функции. В ML используется статическая проверка типов аргументов функций, хотя динамическая проверка типов сделала бы функцию area полиморфной.

Разница между полиморфизмом в ML и общими функциями в Ada заключается в том, что в Ada компилировалось бы несколько различных функций ident, каж-

дая из которых соответствовала бы своему типу аргумента, а в ML достаточно одного экземпляра функции. Вообще говоря, функции в ML, в которых используются равенство (или неравенство) и создание кортежей (tuples) и списков, могут быть полиморфными, так как эти операции не обязательно производятся над фактическими данными.

Аргументы функций ML могут принадлежать любому типу (который определяется статически). Следовательно, мы можем написать полиморфные функции, которые получают параметры и применяют их к объектам данных. Рассмотрим следующие две функции ML: length, которая вычисляет длину списка целых чисел, и printit, которая печатает элементы этого списка в заданном порядке:

```
fun length(nil) =                  0
      | length(a::y) =             1+length(y);
fun printit(nil) =                 print("\n")
      | prx(a::y: int list) =      (print(a):printit(y));
```

Теперь можно вызвать функцию process с аргументами любого типа (единственное требование заключается в том, чтобы они были совместимы), например

```
fun process(f,l) = f l;
```

где f l эквивалентно f(l). Тогда мы можем получить следующее:

```
process(printit,[1,2,3]);
123
process(length,[1,2,3]);
3
```

Реализация. В случае языков со статической проверкой типов, таких как ML и C++, полиморфизм не прибавляет новых сложностей при реализации. Тем не менее с языками, в которых допускается динамический полиморфизм (например, вариант языка LISP под названием Scheme), дело обстоит несколько сложнее, поскольку аргументы полиморфной функции должны быть определены во время выполнения программы.

В качестве примера можно рассмотреть эту ситуацию в языке LISP. Элемент списка в LISP состоит их трех полей, которые мы называли ранее полем типа, полем car и полем cdr (раздел 6.1.7). Поскольку поля car и cdr имеют фиксированный размер, а аргументы полиморфной функции могут быть одного из нескольких различных типов, то для выполнения программы необходимо определить, каков фактический тип аргумента.

Аргумент полиморфной функции может быть представлен одним из двух следующих способов.

1. *Непосредственный дескриптор* встречается тогда, когда значение параметра, передаваемого в функцию, меньше, чем размер фиксированного поля. Например, если функции передается в качестве параметра булево значение, короткое целое число или символ, то используется пространство меньшее, чем позволяет объект фиксированного размера. В таком случае фактическое значение параметра помещается в поле car или cdr, а оставшиеся дополнительные биты в поле используются для сообщения функции о фактическом типе аргумента.

2. *Блоковый дескриптор* используется во всех других случаях. Поля car и cdr будут содержать индикатор типа, указывающий на то, что параметр яв-

ляется блоковым, а остальная часть поля будет содержать адрес памяти фактического объекта (который может располагаться где угодно, например в куче). По этому адресу будет содержаться вся информация о типе этого объекта, например в случае сложного объекта данных — его полная структура.

В качестве примера рассмотрим ситуацию, когда размер поля car или cdr равен 5 байт. Первый байт является идентификатором типа данных, а байты со второго по пятый отведены непосредственно под данные. Полиморфной функции могут быть переданы следующие параметры.

1. *Целое число размером 32 бит.* Первый байт равен 0, что соответствует целому типу параметра, байты со второго по пятый содержат 32-битовое целое число.
2. *Символ размером 8 бит.* Первый байт в данном случае равен 1 (что соответствует символьному типу данных), второй байт содержит фактическое значение аргумента, остальные байты (с третьего по пятый) не используются.
3. *Булево значение размером 1 бит.* Первый байт принимает значение 2, а второй байт равен 0 или 1.
4. *Запись сложной структуры.* Первый байт равен 3, а байты со второго по пятый содержат указатель на запись. *r*-значение, соответствующее этому адресу, предоставляет более полную информацию о фактическом аргументе.

Вычисление такой полиморфной функции будет более медленным, чем для функции в языке со статическим определением типов, например C++, потому что выполняющая программа должна исследовать параметры, прежде чем получить их фактические значения. Тем не менее во многих приложениях выигрыш от использования полиморфных функций перевешивает неэффективность выполнения программ.

7.4. Рекомендуемая литература

В начале 70-х гг. Давид Парнас (David Parnas) внес значительный вклад в разработку инкапсулированных типов данных [86]. Развитие объектно-ориентированного подхода в программировании является предметом рассмотрения в [53]. Конференция ACM «Разработка языков программирования для создания надежного программного обеспечения», состоявшаяся в 1977 г., явилась, вероятно, наиболее глобальным событием подобного рода, где предлагались различные механизмы инкапсуляции в языках Alphard, CLU, Gypsy, Mesa, Euclid и др. [122].

Начало развития объектно-ориентированного подхода к программированию было положено при появлении Smalltalk-72 и его «последователей» [56], хотя признанию этого языка препятствовали проблемы с доступностью его реализаций. Более полное обсуждение вопросов наследования вы найдете в [110]. Полиморфизм и типы в языке ML обсуждаются в [115].

7.5. Задачи и упражнения

1. Определите полиморфную функцию reverse в ML, которая получала бы в качестве аргумента список объектов некоторого типа и возвращала бы эти же объекты, но в обратном порядке.
2. Разработайте набор полиморфных функций (например, push, pop, NewStack, top) в ML, которые позволяли бы создавать стеки, содержащие элементы любого типа.
3. Предположим, что для экземпляров класса object определен метод print. Объясните, как будет выполняться оператор Smalltalk 3 print.
4. Опишите выполнение каждого из следующих выражений Smalltalk:

   а) `2 + 3`

   б) `3 + 2`

   в) `'2' + 3`

   г) `(2 + 3) print`

   д) `(2 < 3) ifTrue:['true' print] ifFalse:['false' print]`
5. Почему использование стандартных файлов включения .h в C не обеспечивает адекватных возможностей абстракции данных для программ на C?
6. Зачем нужны чистые виртуальные классы в C++? Мы могли бы воздержаться от распределения в памяти объектов этого класса и просто использовать подклассы суперкласса.
7. Взгляните еще раз на рис. 7.1, где представлены две модели реализации абстрактных типов данных. Какая модель соответствует Smalltalk, а какая — C++? Как они реализованы?
8. Почему в Smalltalk необходимо иметь различные методы для классов и экземпляров классов? Возможно ли отказаться от одного из этих механизмов и по-прежнему эффективно писать программы на Smalltalk?
9. Рассмотрим следующие объекты: прямоугольник, круг, квадрат, треугольник, многоугольник, прямая, точка, объект, четырехугольник, сфера, трапеция, параллелограмм, шестиугольник, пятиугольник, пирамида, конус. Разработайте для них иерархию классов и определите соответствующие наборы функций, использующие наследование для вычисления объема, периметра и площади поверхности этих объектов (где это возможно).

Глава 8. Управление последовательностью действий

Операции и данные в языках программирования объединяются в программы и совокупности программ при помощи управляющих структур. До сих пор нас интересовали данные и операции сами по себе. Теперь же мы рассмотрим их объединение в законченные, пригодные для выполнения программы. Это объединение имеет два аспекта: во-первых, управление последовательностью выполнения операций, как базовых, так и определенных пользователем, которое мы называем *управлением последовательностью действий* и рассматриваем в этой главе, а во-вторых, управление передачей данных между подпрограммами в пределах одной программы, которое мы называем *управлением данными* и будем рассматривать в следующих двух главах. Такое раздельное рассмотрение удобно ввиду достаточной сложности обеих тем; кроме того, оно поможет понять различия этих двух аспектов языков программирования, которые нередко путают друг с другом.

8.1. Явное и неявное управление последовательностью действий

Структуры управления последовательностью действий удобно разбить на четыре группы.

1. *Выражения* формируют основные строительные блоки для операторов и определяют, каким образом программа управляет данными и изменяет их. Такие свойства языка, как правила приоритета и скобки, устанавливают способы вычисления выражений.
2. *Операторы* или группы операторов, такие как условные операторы и операторы цикла, определяют, каким образом управление передается от одной части программы другой.
3. *Декларативное программирование* — это модель выполнения, независимая от операторов, но тем не менее приводящая к выполнению программы. Примером такой модели может служить модель логического программирования языка Prolog.
4. *Подпрограммы*, такие как вызовы подпрограмм и сопрограммы, формируют способ передачи управления от одной части программы другой. Они подробно обсуждаются в главе 9.

Это разделение не может быть вполне точным. Например, в таких языках, как LISP и APL есть только выражения, но нет операторов, однако и в них используются некие разновидности обычного механизма управления последовательностью выполнения операторов.

Структуры управления последовательностью действий могут быть явными и неявными. *Неявными* структурами (или структурами управления по умолчанию) называются такие, которые, по определению данного языка, действуют во всех случаях, если только программист не изменяет их с помощью какой-либо явной структуры. Например, в большинстве языков по умолчанию используется естественная последовательность выполнения операторов в программе — в том порядке, как они в ней заданы, если только она не изменена явным заданием оператора управления последовательностью действий. Для выражений без скобок обычно также определена иерархия приоритетов выполнения операций. *Явными* структурами управления последовательностью действий называются такие необязательные языковые структуры, которые программист может использовать для изменения неявной последовательности операций, определенной в языке (например, использование скобок внутри выражений или операторов goto и меток).

8.2. Управление последовательностью действий при вычислении арифметических выражений

Рассмотрим формулу для вычисления корней квадратного уравнения:

$$root = \frac{-B \pm \sqrt{B^2 - 4 \times A \times C}}{2 \times A}.$$

Эта на вид простая формула в действительности требует по меньшей мере 15 отдельных операций для ее вычисления (если извлечение квадратного корня считать элементарной операцией и учитывать различные операции получения ссылок на данные). Чтобы запрограммировать эту формулу на стандартном языке ассемблера или машинном языке, потребуется не менее 15 команд, а может быть, и значительно больше. Кроме того, программисту придется следить за каждым из промежуточных результатов и обеспечить для них место в памяти, ему также придется позаботиться об оптимизации, ответив, в частности, на следующие вопросы: можно ли объединить ссылки на два значения A и два значения B, в каком порядке производить операции, чтобы минимизировать объем временной памяти и наилучшим способом использовать возможности аппаратуры? В то же время на языке высокого уровня, например на FORTRAN, формулу для вычисления одного из корней квадратного уравнения можно записать, используя всего одно выражение, практически совпадающее с его математической записью:

```
ROOT=(-B+SQRT(B**2-4*A*C))/(2*A)
```

Эта запись проста и естественна, а вопросами оптимизации и распределения временной памяти занимается языковый процессор, а не программист. Следует отметить, что возможность использования выражений в языках высокого уровня

является одним из их главных преимуществ перед языком ассемблера и машинным языком.

Хотя выражения — это мощный инструмент для представления последовательности операций, они, тем не менее, порождают некоторые новые проблемы. Запись длинных последовательностей команд на машинном языке может оказаться утомительным делом, но зато программист представляет себе точный порядок выполнения команд. А что можно сказать о выражениях? Возьмем выражение для квадратного корня на языке FORTRAN. Корректно ли оно? Откуда мы знаем, что вычитание в этом выражении действительно выполняется *после* вычисления произведения 4*A*C, а не наоборот? Механизмы, управляющие последовательностью действий, которые определяют порядок операций в выражении, на самом деле являются достаточно сложными и тонкими.

8.2.1. Древовидное представление

До сих пор мы рассматривали выражения как одну неделимую единицу и игнорировали синтаксис и семантику, необходимые для вычисления конкретного выражения. При рассмотрении операций внутри выражения аргументы этих операций будем называть *операндами.*

Основным механизмом управления последовательностью действий в выражении является *функциональная композиция*: задается операция и ее операнды, причем операнды могут быть как константами или объектами данных, так и другими операциями, чьи операнды, в свою очередь, могут быть константами, объектами данных или операциями и т. д. — подобных уровней может быть сколько угодно. Функциональная композиция придает выражению характерную структуру дерева, в котором корневой узел соответствует главной операции, узлы, расположенные между корнем и листьями, представляют собой промежуточные операции, а листья являются ссылками на данные (или константы). Например, выражение для вычисления корня квадратного уравнения может быть представлено в виде дерева, изображенного на рис. 8.1 (для обозначения унарного минуса используется символ подчеркивания «_»).

Древовидное представление делает понятнее управляющую структуру выражения. Из такой структуры сразу видно, что результаты ссылок на данные или результаты операций на нижних уровнях дерева служат операндами для операций на более высоких уровнях, и, значит, они должны вычисляться в первую очередь. Но все же древовидное представление определяет порядок выполнения операций не полностью. Так, из рис. 8.1 не ясно, что следует вычислять раньше: -B или B**2; также непонятно, можно ли объединить две ссылки на идентификатор B в одну. К сожалению, как мы увидим позже, при наличии операций с побочными эффектами различные способы решения этих вопросов могут привести к разным результатам. Как правило, в описании языка порядок вычисления выражений определяется на уровне древовидного представления, а определение точного порядка вычислений (например, порядка вычисления -B и B**2) предоставляется разработчику языка. Однако прежде чем рассматривать проблемы, возникающие при определении точного порядка вычислений, следует ознакомиться с различными вариантами синтаксического представления выражений.

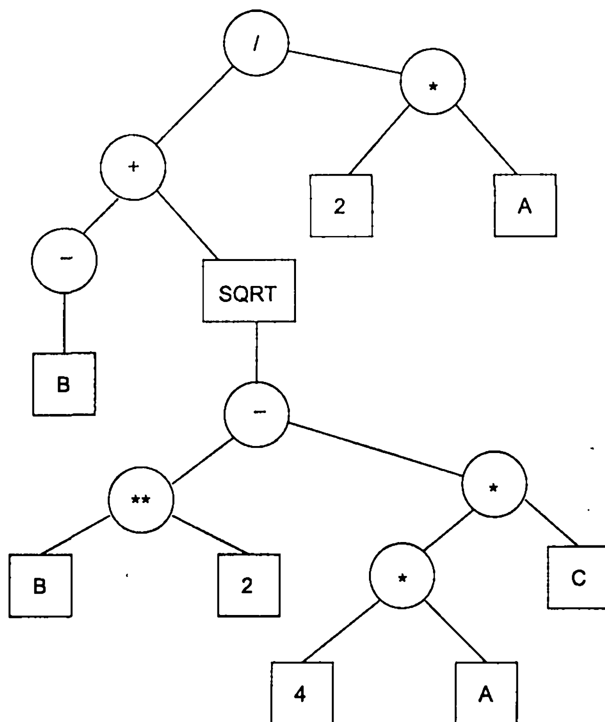


Рис. 8.1. Древовидное представление формулы для нахождения корня квадратного уравнения

Синтаксис выражений

Если мы условимся представлять выражения при помощи деревьев, тогда для использования выражений в программах необходимо осуществить некоторую линеаризацию деревьев (то есть должна быть разработана система обозначений для записи деревьев в виде линейной последовательности символов). Рассмотрим наиболее распространенные способы записи.

Префиксная (польская префиксная) запись. Для обращения к функции в программе обычно имя функции пишется перед ее аргументами, например `f(x, y, z)`. Такой способ записи можно распространить на все операции в выражениях. Так, в префиксной записи сначала записывается символ операции, а затем по порядку, слева направо — операнды. Если операндом является операция, то применяется это же правило. Тогда дерево, изображенное на рис. 8.2, можно представить в виде: `x + a b - c a`. Поскольку + является *бинарной операцией* (то есть требует двух аргументов), то совершенно ясно, что аргументами для + являются `a` и `b`. Аналогично, аргументами для операции - должны быть `c` и `a`. И наконец, аргументами для операции умножения `x` будут член, содержащий +, и член, содержащий –. В такой записи нет никакой неоднозначности, и нет необходимости применять скобки для точного определения последовательности операций при вычислении выражения. Поскольку свободный от скобок способ записи изобрел польский математик Лукашевич (Lukasiewicz), то для обозначения этого способа записи и его производных применяется название *польская запись*.

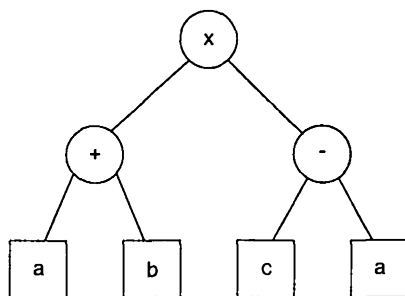


Рис. 8.2. Древовидное представление простого выражения (a + b) × (c – a)

В языке LISP используется один из вариантов этого способа записи, который иногда называют *кембриджской польской записью*. В кембриджской польской записи операция и ее аргументы заключаются в скобки. Выражение принимает вид вложенного множества списков, в котором каждый список начинается с символа операции, за которым следуют списки, представляющие ее операнды. В польской кембриджской записи выражение, отображенное на рис. 8.2, будет выглядеть следующим образом: (*(+ab)(-ca)).

Посмотрим, как будет выглядеть формула для вычисления корня квадратного уравнения в префиксной записи (для обозначения возведения в степень будем использовать символ ↑, для квадратного корня — символ √, а для унарного минуса — символ _):

/ + _ b √ - ↑ b 2 × × 4 a c × 2 a — *польская запись*

(/(+(_b)(√(-(↑ b 2)(×(× 4 a)c))))(× 2 a)) — *кембриджская польская запись*

Постфиксная (суффиксная, или обратная польская) запись. Постфиксная запись похожа на префиксную запись, однако символ, обозначающий операцию, расположен *после* списка операндов. Таким образом, выражение, приведенное на рис. 8.2, будет выглядеть как: a b + c a - ×.

Инфиксная запись. Инфиксная запись наилучшим образом подходит для бинарных (двучленных) операций. При инфиксной записи символ операции пишется между двумя операндами. Поскольку в математике для обозначения основных арифметических и логических операций, а также операций сравнения используется именно инфиксная запись, то она широко применяется для тех же операций и в языках программирования, а в некоторых случаях она распространяется и на другие операции. Дерево выражения, изображенное на рис. 8.2, в инфиксной форме записывается как (a + b) × (c - a). Для операций с количеством аргументов, большим двух (например, в условной тернарной операции языка C), инфиксная запись используется в несколько неуклюжей манере путем применения многократных инфиксных операций: (expr?s_1 : s_2).

Семантика выражений

Каждый из трех способов записи — префиксная, постфиксная и инфиксная — имеет определенные характеристики, полезные при разработке языков программирования. Принципиально они различаются тем, как мы вычисляем значение каждого выражения, представленного в той или иной форме записи. Ниже мы приводим

алгоритмы вычисления (то есть семантику вычисления) для каждого способа записи выражений. Затем мы показываем, что у транслятора имеется возможность выбора: использовать непосредственно этот процесс или, немного изменив его, реализовать вычисление выражения более эффективно.

Префиксное вычисление. С использованием префиксной записи мы можем вычислить любое выражение за один просмотр. Однако при этом необходимо знать количество аргументов для каждой операции. Именно по этой причине необходимо использовать различные символы для бинарного вычитания (-) и унарного минуса (_), чтобы различать их в записи выражения (или же следует использовать кембриджскую польскую запись со скобками).

Помимо того что мы экономим на отсутствии скобок в выражениях, префиксная запись вообще имеет некоторые преимущества при разработке языков программирования.

1. Как отмечалось, обычный вызов функций уже записан в префиксной форме.
2. Префиксную запись можно использовать для представления операций с любым количеством операндов, и поэтому она является наиболее общей. Для записи любого выражения необходимо выучить только одно синтаксическое правило. Например, для написания любого выражения на языке LISP достаточно освоить только кембриджскую польскую запись, и можно сказать, что после этого вам будут известны основные синтаксические правила этого языка.
3. Также префиксную запись относительно легко декодировать механически, поэтому нетрудно осуществить перевод префиксных выражений в простую последовательность кодов.

Этот последний пункт (простота транслирования в последовательность кодов) можно продемонстрировать следующим алгоритмом. Заданное в префиксной форме выражение P, состоящее из операций и операндов, можно вычислить с помощью стека.

1. Если очередной символ в выражении P — символ операции, то помещаем его в вершину стека. Устанавливаем счетчик аргументов так, чтобы он соответствовал количеству операндов, необходимых для вычисления данной операции. (Если количество операндов равняется *n*, то такую операцию называют *n*-арной.)
2. Если следующий элемент — операнд, помещаем его в вершину стека.
3. Если верхние *n* записей в стеке — операнды, необходимые для вычисления верхней *n*-арной операции (например, если сложение + было последней операцией, занесенной в стек, и в него же было добавлено два операнда), то можно применить эту операцию к данным операндам. После выполнения операции заменяем символ операции и *n* ее операндов на результат применения этой операции к соответствующим операндам.

Хотя эта модель вычислений довольно проста, но и здесь существует некоторая сложность — после занесения очередного операнда в стек приходится проверять, имеем ли мы достаточное количество операндов для выполнения соответствую-

щей операции. Чтобы избежать подобной проверки, следует использовать постфиксную запись.

Постфиксное вычисление. Поскольку в постфиксной записи операция следует непосредственно за своими операндами, то при считывании символа операции ее операнды уже известны. Таким образом, вычисление постфиксного выражения P с использованием стека будет выглядеть следующим образом:

1. Если следующий элемент — операнд, помещаем его в стек.
2. Если очередной символ является символом *n*-арной операции, значит, *n* ее аргументов *должны быть* представлены *n* верхними элементами стека. Заменяем эти элементы на результат применения к ним соответствующей операции.

Как видно, стратегия вычислений прямая и легко реализуемая. Фактически в большинстве трансляторов эта стратегия является основой генерации кода для вычисления выражений. В процессе трансляции (см. главу 3) синтаксис выражений часто переводится в постфиксную форму. В этом случае для определения порядка генерируемого кода для вычисления значения выражения генератор кода может использовать приведенный выше алгоритм.

Инфиксное вычисление. Хотя инфиксная запись широко распространена, ее использование в языках программирования создает некоторые особые проблемы.

1. Поскольку инфиксная запись подходит только для бинарных операций, язык не может использовать только одну эту форму записи для выражений, но обязательно должен сочетать ее с префиксной (или постфиксной). Такое смешение делает, соответственно, трансляцию более сложной. Унарные операции и вызовы функций с несколькими аргументами должны быть исключениями из стандартных инфиксных свойств.
2. Когда в выражении появляется более одной инфиксной операции, запись выражения становится неоднозначной, если не использовать скобки.

Этот последний пункт можно проиллюстрировать на примере инфиксного выражения `2 × 3 + 4`. Вы, несомненно, думаете, что значение этого выражения 10, но оно так же легко может оказаться равным 14. На рис. 8.3 показано, почему так может получиться. Когда вы впервые изучали сложение и умножение, вы выучили следующее правило: «Умножение делается прежде, чем сложение» (см. рис. 8.3, *а*). На самом деле, это всего-навсего соглашение, и математика могла бы развиваться так же успешно, если бы был принят обратный порядок действий (см. рис. 8.3, *б*). Конечно, для устранения неоднозначности всегда можно использовать скобки, чтобы явно указать группирование операций и операндов, как, например, `(a × b) + c` или `a × (b + c)`, однако в сложных выражениях большое количество вложенных скобок только сбивает с толку.

По этой причине обычно в языках заранее определены неявные правила вычисления последовательности операций выражения, которые позволяют во многих случаях не использовать скобки. Рассмотрим два примера таких неявных правил.

Иерархия операций (правила старшинства). Все операции, которые могут встретиться в выражениях, выстроены в определенную иерархию в соответствии

с приоритетом их выполнения, или правилом старшинства. В табл. 8.1 приведена иерархия операций языка Ada, которая является типичной и для других языков программирования. Если в выражении встречаются операции с различным приоритетом, то в соответствии с неявными правилами их выполнения первыми выполняются операции с более высоким приоритетом. Таким образом, в выражении a × b + c операция × имеет более высокий приоритет, чем операция +, а значит, и выполняется раньше.

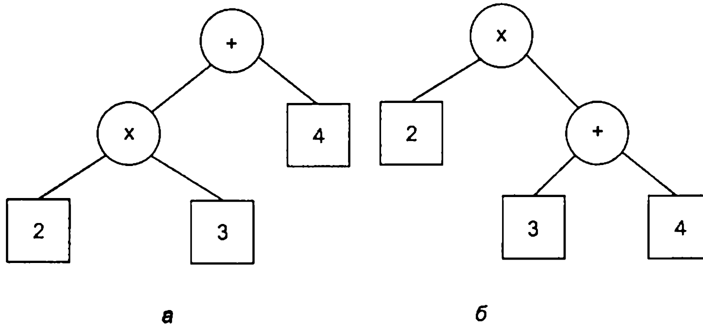


Рис. 8.3. Порядок вычисления операций

Сочетательность. Чтобы полностью определить порядок выполнения операций в выражениях, содержащих операции с одинаковым приоритетом, необходимо дополнительное неявное правило сочетательности. Например, какое вычитание будет производиться первым при вычислении выражения a - b - c? Наиболее часто используется общее неявное правило сочетательности слева направо, то есть a - b - c трактуется как (a - b) - c. (С другой стороны, общепринятое математическое соглашение гласит, что возведение в степень осуществляется справа налево: a ↑ b ↑ c = a ↑ (b ↑ c).)

Таблица 8.1. Иерархия операций языка Ada

Уровень приоритета	Операторы	Операции
Высший приоритет	** abs not	Возведение в степень, абсолютное значение, отрицание
	* / mod rem	Умножение, деление
	+ -	Унарное сложение, вычитание
	+ - &	Бинарное сложение, вычитание
	= ≤ < > ≥	Операции сравнения
Низший приоритет	and or xor	Булевы операции

Для обычных арифметических выражений правила старшинства операций работают корректно, поскольку большинству программистов хорошо известна лежащая в их основе математическая модель семантики выражений. Однако если к языку добавляется новая операция, не из классической математики, правила приоритета перестают работать. C, APL, Smalltalk и Forth являются примерами языков, различными способами оперирующих расширенным набором операций:

✦ *C.* В языке C для расширенного набора операций используется таблица приоритетов, приведенная в табл. 8.2. Для большинства операций имеет место сочетательность «слева направо», за исключением помеченных звездочкой. По большей части распределение приоритетов в языке C вполне логично. В языке Pascal, наоборот, в таблице приоритетов наблюдается странная нелогичность. Булево выражение `a = b | c = d` неверно, поскольку правила приоритетов операций языка Pascal предполагают неестественное группирование, а именно: `a = (b|c) = d`, в то время как программисты обычно имеют в виду совсем другое. При использовании логических выражений в языке Pascal безопаснее всегда использовать скобки.

Таблица 8.2. Уровни приоритета операций языка C

<table>
<tr><th>Уровень приоритета</th><th>Операция</th><th>Краткое описание</th></tr>
<tr><td>17</td><td>Лексемы, a[k], f()</td><td>Константы, индексация, вызов функции</td></tr>
<tr><td></td><td>., -></td><td>Выбор</td></tr>
<tr><td>16</td><td>++, --</td><td>Постфиксный инкремент и декремент</td></tr>
<tr><td>15*</td><td>++, --</td><td>Префиксный инкремент и декремент</td></tr>
<tr><td></td><td>~, -, sizeof</td><td>Поразрядное отрицание, арифметическое отрицание (унарный минус) и получение размера объекта или типа</td></tr>
<tr><td></td><td>!, &, *</td><td>Логическое отрицание, получение адреса и разадресация</td></tr>
<tr><td>14</td><td>(typename)</td><td>Преобразование к типу typename</td></tr>
<tr><td>13</td><td>*, /, %</td><td>Умножение, деление и остаток от деления</td></tr>
<tr><td>12</td><td>+, -</td><td>Сложение и вычитание</td></tr>
<tr><td>11</td><td><<, >></td><td>Сдвиг влево и вправо</td></tr>
<tr><td>10</td><td><, >, <=, >=</td><td>Сравнения</td></tr>
<tr><td>9</td><td>==, !=</td><td>Равенство и неравенство</td></tr>
<tr><td>8</td><td>&</td><td>Поразрядное логическое И</td></tr>
<tr><td>7</td><td>^</td><td>Поразрядное исключающее ИЛИ</td></tr>
<tr><td>6</td><td>|</td><td>Поразрядное логическое ИЛИ</td></tr>
<tr><td>5</td><td>&&</td><td>Логическое И</td></tr>
<tr><td>4</td><td>||</td><td>Логическое ИЛИ</td></tr>
<tr><td>3*</td><td>?:</td><td>Условная операция</td></tr>
<tr><td>2*</td><td>=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=</td><td>Выполнение операции с присваиванием</td></tr>
<tr><td>1</td><td>,</td><td>Последовательное вычисление</td></tr>
</table>

* Сочетательность операции справа налево.

✦ *APL.* Базовые операции этого языка разработаны для выполнения действий с векторами и массивами. Почти все операции языка APL новые в том смысле, что при первоначальном их изучении они кажутся странными любому программисту, работавшему с другими языками, но не APL. Назначение каких-либо приоритетов операциям APL выглядело бы ис-

кусственным, поэтому данный язык не имеет приоритетов и все выражения в нем вычисляются справа налево. В большинстве программ на APL такой подход обоснован, за исключением того, что некоторые типичные выражения выглядят несколько неестественно. Например, выражение a - b - c, где a, b, c — целые, вычисляется как a - (b - c), что означало бы a - b + c на таких языках, как ML, FORTRAN или C. APL кратко описан в обзоре языка 8.1.

Обзор языка 8.1. APL

Возможности. Сильной стороной языка APL является обработка массивов. К векторам и массивам применимы все арифметические операции, и, кроме того, существуют дополнительные операции, например создание вектора, все элементы которого имеют заданную величину. Поскольку многие из этих векторных операций не являются интуитивно понятными, в языке APL не существует приоритетов операций и все операторы выполняются справа налево — странная особенность, к которой надо привыкнуть.

История. Язык APL был разработан Кеном Иверсоном (Ken Iverson) в начале 60-х гг. как система обозначений для описания вычислений. Позднее он использовался как язык машинной архитектуры, в котором поведение конкретных команд можно было легко описать в виде векторных операций APL. Третьей фазой развития APL стала его реализация для IBM 360 в конце 60-х гг. Вследствие краткости выражений на APL, у этого языка появились немногочисленные, но преданные сторонники, которые гордились тем, что легко разрабатывали сложные программы.

Пример. Вычисление суммы элементов массива

1	k ←	Прочитать размер массива в k
2	A ←	Прочитать первый элемент массива
3	A ← A,	Прочитать следующий элемент массива, расширить массив
4	→3×ιk>ρA	Размер массива A (операция ρ A) сравнивается с k и выдает 0 или 1. Генерирует вектор из 0 или 1 (операция ι) и переходит (операция →) к оператору 3, если результат 1 (то есть суммирование еще не закончено)
5	← +/A	Добавляет элементы массива (+/) и выводит сумму

Ссылка. A. D. Falkoff and K. E. Iverson "The evolution of APL", *ACM History of programming Languages Conference*, Los Angeles, CA (June 1978) (*SIG-PLAN Notices (13)8 [August 1978]*), 47–54.

✦ *Smalltalk.* В языке Smalltalk используется та же модель, что и в APL. Поскольку этот язык предназначен для построения функций (методов), обеспечивающих выполнение заданных функциональных возможностей, то никогда не известно, каким приоритетом будет обладать новая функция. Поэтому обычно приоритеты отсутствуют, а выражения вычисляются слева направо. (Более подробно о языке Smalltalk можно прочитать в разделе 7.2.5.)

- *Forth.* Язык Forth был разработан для функционирования на компьютерах, управляющих процессами в режиме реального времени. Такие машины создавались в шестидесятые годы и представляли собой дорогие мини-компьютеры. Поскольку память стоила дорого, любой используемый на них язык должен был быть компактным, программы на нем должны были легко транслироваться и эффективно выполняться. Как отмечалось ранее, вычисление постфиксных выражений осуществляется достаточно легко. В языке Forth структура исполняемой программы представляла собой стек, а синтаксис языка был строго постфиксным. Это позволяло устанавливать трансляторы во многие приложения без больших затрат и усилий. При использовании строго постфиксной записи не возникал вопрос о приоритетах операций. Язык Forth кратко описан в обзоре языка 8.2. На сегодняшний день этот язык сохранился в виде основы для языка Postscript (раздел 12.1).

Обзор языка 8.2. Forth

Возможности. Постфиксный язык позволяет разработать эффективную модель выполнения, даже несмотря на то, что, как правило, такой язык является интерпретируемым. Система работает с использованием двух стеков — стека возврата из подпрограммы и стека вычисления выражения. Небольшая модель выполнения программ, написанных на таком языке, позволяет использовать его для малых встроенных компьютеров.

История. Язык Forth был разработан Чарльзом Муром (Charles Moore) около 1970 г. Название этого языка возникло как сокращение до пяти букв первого слова в полном названии программы его разработки «Fourth Generation Programming Language». Этот язык в 70-е гг. служил заменой языку FORTRAN для мини-компьютеров, у которых память являлась дефицитным ресурсом в силу ее дороговизны, а единственное устройство ввода-вывода, как правило, было очень медленным и громоздким печатающим устройством на бумажной ленте. Наличие резидентного транслятора-интерпретатора облегчало разработку программ для подобных встроенных систем. Язык Forth не пользовался большим успехом, но он до сих пор существует в качестве основного «движущего механизма» языка Postscript (глава 12).

Пример. Написать программу, которая вычисляет сумму $1^2 + 2^2 + ... + 9^2 + 10^2$:

```
(Обозначения: a. b. c – стек выражения. Элемент c – на вершине стека)
: SQR DUP * : (Определяет квадратный корень как: n ⇒ n. n ⇒ (n*n))
: DOSUM SWAP 1 + SWAP OVER SQR + :     (N. S ⇒ N+1. S+(N+1)²)
                                        (N. S ⇒ S. N ⇒ S. (N+1) ⇒ (N+1). S ⇒
                                        (N+1). S. (N+1) ⇒ (N+1). S. (N+1)² ⇒
                                        (N+1). S+(N+1)²)
3 6 DOSUM . . 22 4 ok                  (Операция точка (.) печатает верхний элемент стека.
Вывод 22 = 4² + 6)
0 0 10 0 DO DOSUM LOOP . 3B5 ok    (Применяем DOSUM от 0 до 9 (останавливаемся при 10))
```

Ссылка. E. Rather, D. Colburn, and C. Moore "The evolution of Forth", *ACM History of Programming Languages Conference II*, Cambridge, MA (April 1993) *(SIGPLAN Notices (28)3 [March 1993])*, 177–199.

Каждый из способов записи выражений, перечисленных выше, имеет свои собственные недостатки. Инфиксная запись со своими неявными правилами при-

оритетов и сочетательностью операций, а также с явным использованием (при необходимости) скобок дает довольно естественное представление для большинства арифметических и логических выражений и выражений сравнения. Однако необходимость использования сложных неявных правил и префиксной (или другой) записи для операций, отличных от бинарных, усложняет трансляцию таких выражений. Инфиксная запись, лишенная неявных правил, является громоздкой из-за необходимости использовать большое количество скобок. Однако как в кембриджской польской, так и в обычной математической префиксной записи проблема со скобками все же присутствует. В польской записи вообще не используются скобки, но необходимо заранее знать количество операндов для каждой операции, а это условие обычно трудно выполнить (особенно если речь идет о расширении языка операциями, которые определены программистами). Наконец, отсутствие каких-либо структурирующих подсказок затрудняет чтение сложных выражений, записанных с помощью польской записи. Как префиксная, так и постфиксная запись имеет преимущество при записи операций с различным числом операндов.

8.2.2. Представление выражений во время выполнения программы

Ранее были приведены алгоритмы для понимания семантики выражений, записанных во всех трех формах. Однако если сначала перевести каждое выражение в его древовидное представление, у транслятора появится возможность выбора эффективного способа вычисления выражения. Когда выражение может быть представлено в инфиксной форме, неявные правила приоритетов и сочетаемости операций позволяют на первой стадии трансляции установить основную древовидную управляющую структуру выражения. На второй, необязательной стадии принимаются уточненные решения, связанные с порядком вычисления, включая оптимизацию этого процесса.

Выражения в своей исходной инфиксной форме в тексте программы трудны для декодирования, поэтому их принято транслировать в выполняемую форму, легко декодируемую в процессе выполнения программы. Среди используемых вариантов наиболее важны следующие.

1. *Последовательность машинных команд.* Выражение транслируется прямо в машинные коды, две стадии трансляции объединяются в одну. Порядок команд отражает структуру управления последовательностью действий исходного выражения. На традиционных компьютерах для хранения промежуточных результатов такие последовательности машинных команд должны использовать явно выделяемую временную память. Конечно, представление в виде машинных кодов допускает использование аппаратного интерпретатора, что обеспечивает очень высокую скорость выполнения.

2. *Древовидные структуры.* Выражение может быть преобразовано непосредственно в его естественное древовидное структурное представление (первый этап) с помощью программного интерпретатора. Его вычисление (второй этап) может быть затем осуществлено простым обходом дерева. Этот

метод является основным для программно интерпретируемого языка LISP, в котором во время выполнения вся программа представляется в виде древовидных структур.

3. *Префиксная или постфиксная форма.* Выражения, представленные в префиксной или постфиксной форме, можно вычислять с помощью приведенного выше простого алгоритма интерпретации (причем оба этапа осуществляются за один шаг). В некоторых компьютерах, основанных на стековой организации, машинный код, по существу, представлен в постфиксной форме. Префиксное представление является выполняемой формой программы во многих реализациях языка SNOBOL4. Выполнение осуществляется сканированием выражения слева направо, причем каждая операция рекурсивно вызывает интерпретатор для вычисления своих операндов.

Вычисление выражения по его представлению в виде дерева

Хотя трансляция выражений в программах в их представления в виде дерева иногда и вызывает затруднения, но на самом деле основная процедура трансляции проста. Вторая стадия, во время которой дерево преобразуется в выполняемую последовательность элементарных операций, включает решение большинства тонких вопросов, касающихся порядка вычисления выражения. Мы не собираемся здесь изучать сами алгоритмы генерации выполняемого кода на основе представления выражения в виде дерева, а всего лишь хотим рассмотреть те проблемы, касающиеся порядка вычисления выражений, которые возникают при точном определении генерируемого кода.

Проблема 1. Унифицированные правила вычислений. При вычислении выражения или генерации кода для его вычисления кажется очевидным применение следующего унифицированного правила вычислений. Для каждого узла, представляющего операцию в дереве выражения, сначала вычисляются все его операнды (или генерируется код для их вычисления), а затем к вычисленным операндам применяется операция (или генерируется код для применения операции). Поскольку всегда сначала вычисляются операнды, мы называем это правилом *активного* вычисления. Точный порядок выполнения этих вычислений не должен иметь значения, поэтому можно выбрать порядок вычисления операндов или независимых операций из соображений оптимизации использования временной памяти или каких-либо иных аппаратных характеристик компьютера. Этому правилу вычислений удовлетворяет любой из приведенных ниже порядков вычисления выражения `(a+b) × (c-a)`, древовидная структура которого изображена на рис. 8.2.

Порядок 1. Сначала вычисляем `a + b`.

1. Извлекаем из памяти *r*-значение переменной `a`.
2. Извлекаем из памяти *r*-значение переменной `b`.
3. Складываем `a` и `b`, получая `d`.
4. Извлекаем из памяти *r*-значение переменной `c`.
5. Вычитаем `a` из `c`, получая `e`.
6. Перемножаем `d` и `e`, получая `f`, являющееся *r*-значением выражения.

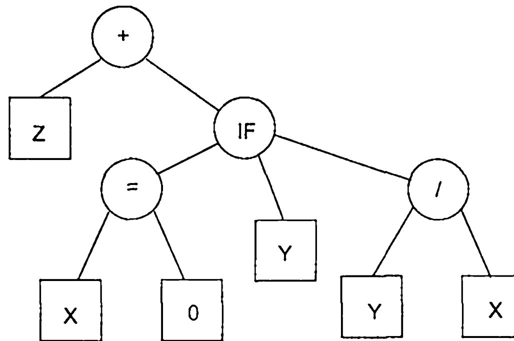


Рис. 8.4. Выражение, содержащее условную операцию

Порядок 2. Вычисляем операнды до выполнения любой операции.

1. Извлекаем из памяти *r*-значение переменной c.
2. Извлекаем из памяти *r*-значение переменной b.
3. Извлекаем из памяти *r*-значение переменной a.
4. Вычитаем a из c, получая e.
5. Складываем a и b, получая d.
6. Перемножаем d и e, получая f.

Все это выглядит совершенно естественно, и было бы заманчиво всегда применять это унифицированное правило вычислений. К сожалению, оно не всегда применимо. Наилучшим примером являются выражения, содержащие условные операции. Например, следующее выражение на языке C: `Z + (Y = 0 ? X : X/Y)`, содержит встроенный оператор `if`, который вычисляет X/Y, если Y не равняется 0. Может возникнуть желание рассматривать эту условную операцию просто как операцию с необычным синтаксисом и тремя операндами, как показано на рис. 8.4. На самом деле, например, в языке LISP так и поступают, используя кембриджскую польскую запись для условных операций точно так же, как и для любых других. Но здесь и возникает проблема с унифицированным правилом вычислений. Если мы применим это правило и вычислим сначала операнды условной операции, изображенной на рис. 8.4, мы сделаем как раз то, во избежание чего и понадобилось условие, а именно поделим X на Y, даже если Y будет равен 0. Ясно, что в данном случае не следует вычислять все операнды перед применением операции. Наоборот, операнды условной операции (или, по крайней мере, последние два) нужно оставить *не вычисленными* и предоставить самой операции определять порядок вычисления.

Проблема с условными операциями наводит на мысль, что, вероятно, лучше будет использовать другое унифицированное правило вычислений, обычно называемое правилом *ленивого* вычисления: *никогда* не вычислять операнды перед применением операции. Наоборот, всегда оставлять операнды не вычисленными и позволять самой операции решать, необходимо ли их вычисление до выполнения операции. Это правило работает во всех случаях и теоретически могло бы использоваться как универсальное правило. Однако в большинстве случаев его реализация оказывается непрактичной. Как, например, смоделировать передачу операции еще не вычисленных операндов? Это потребует выполнения значительного объе-

ма программного моделирования. Зачастую такой подход к вычислению выражений используется в интерпретируемых языках типа LISP и Prolog, но для языков численных расчетов, например для C и FORTRAN, использование *ленивого* правила будет связано с непомерно высокими издержками.

Два предложенных выше унифицированных правила вычислений — *активное* и *ленивое* — соответствуют двум распространенным методам передачи параметров в подпрограммы: *по значению* и *по имени* соответственно. Более подробно эти концепции и способы их реализации описаны в следующей главе, где рассматривается передача параметров. Здесь же для наших целей достаточно отметить, что ни одно простое унифицированное правило вычисления выражений (или генерация кода для их вычисления) не является удовлетворительным. В конкретных реализациях обычно можно обнаружить некоторую комбинацию этих двух методов. В языке LISP, например, функции (операции) делятся на две категории в зависимости от того, получают они вычисленные или не вычисленные операнды. В языке SNOBOL4 определяемые программистом операции (подпрограммы) всегда получают вычисленные операнды, в то время как элементарные операции, определенные в языке, получают не вычисленные операнды. Элементарные операции языка ALGOL получают вычисленные операнды, при этом условные операции моделируются встраиваемой последовательностью машинных команд, а подпрограммы, определяемые программистами, могут получать как вычисленные, так и не вычисленные операнды.

Проблема 2. Побочные эффекты. Использование в выражениях операций с побочными эффектами — основной предмет продолжающегося до сих пор спора в области принципов разработки языков программирования. Рассмотрим выражение:

```
a × fun(x) + a
```

Прежде чем выполнять умножение, следует извлечь из памяти *r*-значение переменной a и вычислить fun(x). Для сложения требуется значение a и результат операции умножения. Очевидно, что желательно извлечь значение a из памяти только один раз, а затем в процессе вычисления использовать его в двух местах. Более того, должно быть безразлично, когда именно вычисляется fun(x): до извлечения значения a из памяти или после. Однако если при вычислении fun вследствие побочного эффекта изменяется значение a, то точный порядок вычисления становится существенным. Например, если вначале a имело значение 1, а fun(x) возвращает значение 3 и также изменяет значение a на 2, тогда для различных порядков вычисления получаются различные результаты.

1. *Вычисление в порядке слева направо дает результат:* 1 × 3 + 2 = 5.
2. *Вычисление в порядке справа налево дает результат:* 1 + 3 × 2 = 7.
3. a *вычисляется только один раз:* 1 × 3 + 1 = 4.
4. *Значение* fun(x) *вычисляется до вычисления* a: 3 × 2 + 2 = 8.

Все эти значения корректны с точки зрения синтаксиса языка, и выбор одного из них определяется порядком вычисления компонентов выражения.

По вопросу об использовании побочных эффектов возникло две точки зрения. Согласно одной из них, побочных эффектов в выражениях допускать нельзя. Для этого следует либо вообще исключить использование в выражениях функций, вызывающих побочные эффекты, либо просто считать неопределенным значение

любого выражения, в котором побочные эффекты могут повлиять на его значение (например, значение выражения в предыдущем примере). Другая точка зрения состоит в том, что побочные эффекты могут присутствовать. Тогда из определения языка должен быть совершенно ясен порядок вычисления компонентов выражения, чтобы программист мог правильно использовать побочные эффекты в своей программе. Однако в этом случае многие виды оптимизации становятся невозможными. В определениях многих языков этот вопрос, к сожалению, полностью игнорируется, в результате чего в разных реализациях языка даются различные интерпретации, противоречащие друг другу.

Как правило, допускается возможность побочных эффектов для операторов. Например, операция присваивания всегда производит побочный эффект — изменяет значение переменной или элемента структуры данных. Ясно также, что в последовательности операторов побочный эффект одного оператора может влиять на исходные данные другого. Проблема заключается в следующем: следует ли допускать этот вид взаимозависимости через побочные эффекты ниже уровня операторов в выражениях. Если не допускать, то необходимо определить порядок вычисления в выражениях только до уровня представления в виде дерева; с точки зрения программиста вычисление выражений происходит без всяких «трюков», и у транслятора есть возможность оптимизации последовательности вычисления выражения. Однако если оптимизация не является приоритетной задачей, то часто бывает целесообразно допустить побочные эффекты и полностью определить порядок вычисления выражений. В этом случае у нас почти не остается оснований различать в языке операторы и выражения. На самом деле в некоторых языках, таких как LISP и APL, различие между выражениями и операторами полностью или частично устранено. Это заметно упрощает работу программиста. В целом, не существует какого-либо одного преобладающего взгляда на побочные эффекты, у каждого подхода есть свои сторонники.

Проблема 3. Ошибки. Особый вид побочного эффекта встречается в случае неуспешного завершения операции и, соответственно, генерирования состояния ошибки. В отличие от обычных побочных эффектов, встречающихся, как правило, только в функциях, определенных программистом, состояние ошибки (переполнение, деление на ноль) может возникать при выполнении многих элементарных операций. Несмотря на то что смысл и даже само появление подобной ошибки могут зависеть от порядка вычисления компонентов выражения, запрещать такие побочные эффекты нежелательно. В таких ситуациях программисту может понадобиться возможность точного управления порядком вычисления выражения, хотя требования оптимизации могут препятствовать этому. Разрешение этих трудностей по своей сути зависит от ситуации и может изменяться в зависимости от используемых языка и реализации.

Проблема 4. Вычисление булевых выражений по укороченной схеме. При написании программ для объединения выражений сравнения обычно используются логические (булевы) операции И (&& в языке С) и ИЛИ (|| в языке С), как, например, в следующих операторах языка С:

```
if ((A == 0) || (B/A > C)) {…}
```

и

```
while ((I <= UB) && (V[I] > C)) {…}
```

В обоих этих выражениях вычисление второго операнда булевой операции может привести к возникновению ошибки (деление на ноль, выход индекса из диапазона его значений); первый операнд и добавлен именно для того, чтобы ошибка не произошла. В языке C, если левый операнд булева выражения в первом примере вычисляется равным значению *истина* и *ложь* — во втором, то второй операнд вообще никогда не вычисляется. Логически это имеет смысл, поскольку ясно, что значение булева выражения α || β будет *истина*, если α принимает значение *истина*, и аналогично значение выражения α && β будет *ложь*, если значение α — *ложь*. К сожалению, проблема унифицированного вычисления присутствует и здесь. В большинстве языков оба операнда вычисляются еще до вычисления булевой операции. Многие ошибки программирования возникают из-за неоправданного расчета на то, что вычисленное значение левого операнда булевой операции может отменить вычисление правого операнда, если значение всего выражения может быть определено из значения только левого операнда. Для разрешения данной проблемы в языке Ada, в дополнение к обычным булевым операциям and и or, которые не вычисляются по укороченной схеме, введены две специальных булевых операции, and then и or else, которые явным образом осуществляют вычисление по укороченной схеме. Например, на языке Ada выражение в условном операторе

```
if (A = 0) or else (B/A > C) then ...
```

не может вызвать ошибки деления на ноль, поскольку если A = 0, то вычисление всего выражения прекращается и его значение принимается равным *истина*.

8.3. Управление последовательностью выполнения операторов

В этом разделе будут рассмотрены основные механизмы, используемые для управления последовательностью выполнения отдельных операторов. Структуры управления большой группой операторов, относящиеся к программам и подпрограммам, будут описаны в следующей главе.

8.3.1. Базовые операторы

Результаты выполнения программы определяются ее *базовыми операторами*, которые применяют операции к объектам данных. Примерами таких базовых операторов могут служить операторы присваивания, вызовы подпрограмм и операторы ввода-вывода. Как описывалось в предыдущем разделе, последовательность выполнения операций внутри базового оператора может определяться используемыми выражениями. Однако в нашем случае каждый базовый оператор можно рассматривать как единое целое, представляющее отдельный шаг в вычислениях.

Присваивание значений объектам данных

Изменение состояния вычислений путем присваивания значений объектам данных является основным механизмом, который влияет на состояние вычислений, определяемых программой. Существует несколько видов таких операторов.

Оператор присваивания. Краткое обсуждение операторов присваивания можно найти в разделе 5.1.5. Основной задачей присваивания является присвоение *l*-значению объекта данных (то есть области памяти, выделенной для объекта) *r*-значения (то есть значения объекта данных) некоторого выражения. Присваивание является центральной операцией, определенной для каждого элементарного типа данных. Синтаксис явного присваивания в разных языках сильно различается.

A := B	Pascal, Ada
A = B	C, FORTRAN, PL/1, Prolog, ML, SNOBOL4
MOVE B TO A	COBOL
A ← B	APL
(SETQ A B)	LISP

В языке C присваивание является просто операцией, так что можно написать: `c = b = 1`; согласно порядку приоритетов, представленному в табл. 8.2, данная запись означает `(c = (b = 1))` и ей соответствует следующий вычислительный процесс:

1. Переменной `b` присваивается значение 1.
2. Выражение `(b = 1)` возвращает значение 1.
3. Переменной `c` присваивается значение 1.

Чаще, однако, присваивание рассматривается как отдельный оператор. Поскольку операция присваивания языка Pascal не возвращает никакого явного результата, мы используем ее только на уровне операторов в явном операторе присваивания:

```
X := B + 2*C;
Y := A + X;
```

В большинстве языков имеется единственная операция присваивания. Однако в языке C их несколько:

A = B	Присваивает *r*-значение переменной B *l*-значению переменной A, возвращает *r*-значение переменной A.
A+= B (-=)	Увеличивает (или уменьшает) значение переменной A на значение переменной B (A = A + B или A = A - B), возвращает новое значение.
++A (--A)	Увеличивает (или уменьшает) значение переменной A на 1 и возвращает новое значение (например, A = A + 1, возвращает *r*-значение A).
A++ (A--)	Возвращает значение переменной A, затем увеличивает (или уменьшает) ее значение на 1 (сначала возвращает *r*-значение A, затем A = A + 1).

Хотя все эти операции похожи друг на друга, но каждая имеет немного отличную от других семантику и может приводить к разным эффектам в различных ситуациях. Поскольку основной операцией присваивания является присваивание *r*-значения одного выражения *l*-значению другого выражения, можно достичь большей гибкости, если иметь операции, воздействующие на *r*-значения и *l*-значения переменных. Например, в языке C унарная операция разадресации * является операцией, которая заставляет *r*-значение переменной вести себя так, как будто

оно является *l*-значением. А унарная операция взятия адреса &, наоборот, преобразует *l*-значение в *r*-значение. Рассмотрим пример:

```
int i. *p:
p = &i:
*p = 7:
```

Здесь происходит следующее:

1) объявляется целая переменная i;
2) p объявляется как указатель на целую переменную;
3) p устанавливается как указатель на i, то есть *l*-значение переменной i преобразуется в *r*-значение (&i) и это *r*-значение сохраняется как *r*-значение p;
4) *r*-значение переменной p преобразуется в *l*-значение (*p), но это — *l*-значение переменной i, следовательно, *r*-значение переменной i устанавливается равным 7.

Ниже приведен более полный пример программы на языке С:

```
main()
{int *p. *q. i. j:                              /* p и q - указатели на целые */
int **qq:                                       /* qq - указатель на указатель на целое */
i=1: j=2: printf("I=%d: J=%d:\n".i.j):          /* печатает i.j */
p = & i:                                        /* p= l-значение переменной i */
q = & j:                                        /* q= l-значение переменной j */
*p =*q: printf("I=%d: J=%d:\n".i.j):            /* то же. что i = j */
qq = & p:                                       /* qq указывает на p */
**qq = 7: printf("I=%d: J=%d:\n".i.j):}         /* то же. что i = 7 */
```

Вывод из этой программы следующий:

```
I=1: J=2:
I=2: J=2:
I=7: J=2:
```

Оператор ввода. Большинство языков программирования включают операторную форму для чтения вводимых пользователем данных с терминала, из файлов или из линии связи. Эти операторы также изменяют значение переменных через операции присваивания. Обычно синтаксис таких операторов выглядит как read(file. data). В языке С вызов функции printf приводит к записи файла в буферную переменную. В языке Perl простое упоминание входного файла вызывает операцию чтения (например, $X=<STDIN> вызывает присваивание переменной $X очередной строки ввода).

Другие операторы присваивания. Передача параметров (раздел 9.3) обычно определяется как присваивание значения аргумента формальному параметру. Также в языках программирования можно найти различные формы неявного присваивания (например, в языке SNOBOL4 каждая ссылка на переменную INPUT приводит к присваиванию этой переменной нового значения), а сопоставление с целью в языке Prolog (например, резолюция) также вызывает неявное присваивание значений переменным. Часто можно присвоить начальное значение переменной при ее объявлении.

Формы управления последовательностью действий на уровне операторов

Обычно выделяют три формы управления последовательностью действий на уровне операторов:

- *Композиция.* Операторы могут быть представлены в виде текстовой последовательности, так что всякий раз, когда выполняется большая программная структура, содержащая эту последовательность, операторы последовательности выполняются в надлежащем порядке.
- *Ветвление.* Две последовательности операторов могут быть альтернативными, так что при выполнении большей программной структуры, содержащей обе эти последовательности, выполняется только одна из них, но не обе вместе.
- *Повторение.* Последовательность операторов может выполняться неоднократно, 0 или более раз (ноль означает, что выполнение может быть вообще опущено), когда бы ни выполнялась большая программная структура, содержащая эту последовательность операторов.

Для достижения желаемого эффекта при конструировании программы мы объединяем элементарные операторы, производящие вычисления, в определенные последовательности с помощью многократного использования композиции, ветвления и повторения. Часто используются вариации каждой из этих основных форм управления, соответствующие конкретной цели. Например, часто приходится выбирать из нескольких, а не из двух вариантов последовательностей. Обычно в языке программирования предусматриваются различные структуры управления последовательностью действий, предназначенные для упрощения выражения подобных форм управления.

Явное управление последовательностью операторов

Ранние языки программирования разрабатывались для существовавших в то время конкретных вычислительных машин, на которых и предполагалось выполнять программы, написанные на этих языках. Поскольку память машин представлялась в виде определенных областей, ранние языки (например, FORTRAN, ALGOL) моделировали их с помощью простых типов данных, непосредственно переводимых в машинные объекты (например, тип переменных `float` языка С или `real` в языке FORTRAN переводился в аппаратный тип с плавающей точкой, а тип `int` языка С — в аппаратный целочисленный тип), и простых операторов, состоящих из меток и ветвей. Передача управления чаще всего указывается с помощью `goto` — оператора перехода на оператор с заданной меткой.

Во многих языках часто присутствует два вида оператора `goto`:

- *Безусловный* `goto`. Если в последовательности операторов встречается оператор безусловного перехода `goto`, например:

  ```
  goto NEXT
  ```

  то управление передается оператору, помеченному меткой `NEXT`, а оператор, непосредственно следующий за оператором `goto`, не выполняется.
- *Условный* `goto`. Если в последовательности операторов встречается оператор условного перехода `goto`, например:

  ```
  if A = 0 then goto NEXT
  ```

  то управление передается оператору, помеченному меткой `NEXT`, только если выполняется заданное условие.

Хотя оператор goto — вполне обычный оператор, за последние 30 лет ему пришлось пережить нелегкие времена. Его использование при написании программ подвергалось постоянной критике. Как будет показано ниже, оператор goto нарушает принцип структурного проектирования программы. Например, большая часть формального моделирования, приводящего к аксиоматически корректной модели разработки программы (раздел 4.2.4), зависит от рациональной структуры управления для программы. Фактически, используя аксиоматическую модель, описанную в упомянутом разделе, не так просто включить в арсенал разработки оператор goto.

Более важно то, что, как было показано, оператор goto является на самом деле излишним. (См. структурную теорему в этой же главе, раздел 8.3.3.) Любую программу можно написать без оператора goto, и при обучении языкам C или Pascal большинству студентов даже не сообщают о его существовании в этих языках.

Оператор break. Некоторые языки, такие как C, включают оператор break как форму явного структурированного управления выполнением операторов. Обычно оператор break передает управление оператору, непосредственно следующему за структурой управления, в которой он сам содержится. Так, в языке C оператор break вызывает немедленный выход из операторов while, for, switch, в которых он встречается. Использование оператора break продолжает порождать структуру управления с одним входом и одним выходом, которая позволяет разрабатывать формальные свойства программы (рис. 8.5).

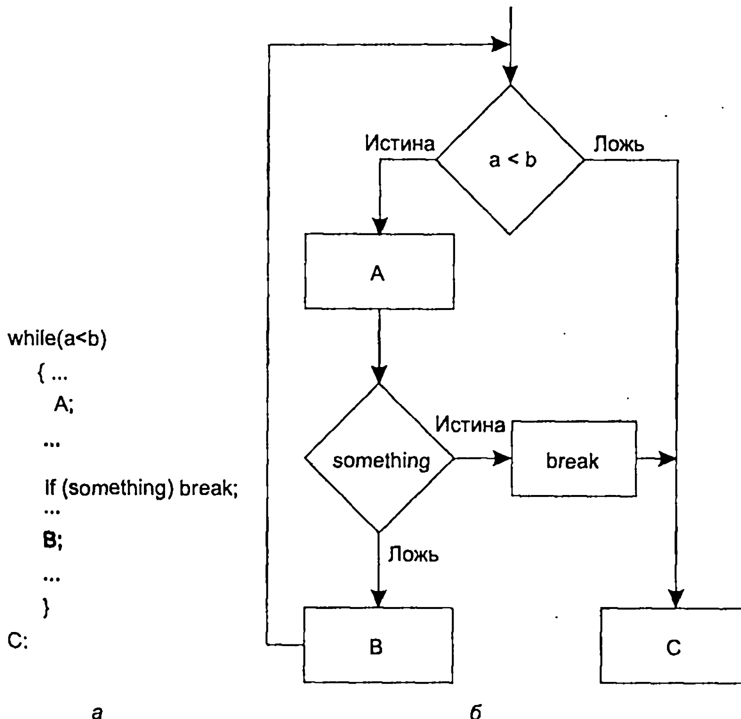


Рис. 8.5. Структурный оператор break: *а* — синтаксис; *б* — выполнение оператора

Также язык С включает в себя родственный оператору break оператор continue, который завершает выполнение текущей итерации цикла в операторах while и for и переводит управление в конец тела цикла. Таким образом, оператор break прерывает цикл, в то время как оператор continue передает управление следующей итерации.

Однако использование оператора break языка С (и связанного с ним оператора switch) иногда является причиной нестабильной работы программ на С. Если нет оператора break, то от одного ключа оператора switch управление переходит к следующему. Также может возникнуть путаница, из какой именно конструкции при наличии оператора break следует выйти. В начале 60-х гг. в США произошел большой сбой в работе телефонной сети фактически из-за неправильного размещения оператора break.

Парадигма структурного программирования

Хотя в большинстве языков определены метки и операторы goto, уже в 70-е гг. имели место серьезные споры по поводу их применения. В некоторых новых языках (например, ML) оператор goto отсутствует. Перечисленные ниже недостатки использования оператора goto сильно перевешивают любое из его достоинств.

Отсутствие иерархической структуры программы. Правильность работы программы гораздо важнее эффективности ее выполнения. При разработке языка программирования должно выполняться это требование.

Концепция *управляющей структуры с одним входом и одним выходом* обеспечивает более понятную структуру программы. В программе, состоящей из большого количества операторов, трудно разобраться, если нет иерархической организации операторов в группы, так чтобы каждая группа представляла собой один концептуальный элемент программы. Каждая из этих групп организуется как несколько подгрупп, использующих одну из форм управления и т. д. В структуре программы такой вид ее иерархической организации очень существенен, так как позволяет программисту понять, как все части программы согласованы друг с другом.

Порядок операторов в тексте программы не будет соответствовать реальной последовательности их выполнения. С помощью операторов goto легко писать программы, в которых управление перескакивает между различными последовательностями операторов беспорядочным образом. Поэтому та последовательность, в которой операторы появляются в программе, не отражает последовательность их выполнения, а значит, такие программы трудны для понимания. Программы с большим количеством нерациональных передач управления назад и вперед часто называются *кодом «спагетти»* (см. рис. 1.2).

Группы операторов могут служить различным целям. Программа более проста для понимания, если каждая группа операторов в пределах всей структуры программы служит единственной цели (то есть вычисляет ясно определенную отдельную часть всего целого вычисления). Часто бывает, что разные группы операторов могут содержать несколько одинаковых операторов. Если использовать оператор goto, можно объединить эти группы таким образом, чтобы писать одинаковые операторы только один раз и передавать управление этому общему набору операто-

ров во время выполнения каждой группы операторов. Такой подход затрудняет понимание программы.

Структурное программирование. Этот термин используется для обозначения методологии разработки программ, в которой особое внимание уделяется:

1) иерархической разработке программных структур, использующей только простые формы управления: композицию, ветвление и повторение, — описанные ранее в этой же главе;
2) представлению иерархической структуры программы непосредственно в ее тексте, использующему «структурированные» операторы управления, описываемые далее в этой же главе;
3) тексту программы, в котором последовательность появления операторов должна соответствовать последовательности их выполнения;
4) использованию групп операторов, предназначенных для вычисления одной цели, даже если при этом операторы придется просто копировать.

Обычно, если программа написана в соответствии с правилами структурного программирования, то ее легче понять, отладить, проверить на корректность, а позже модифицировать и снова проверить на корректность. В разделе 8.3.3 приведена модель структур управления, называемых *базовыми структурами управления*, которая помогает формально определить то, что мы понимаем под структурированной программой.

8.3.2. Структурированное управление последовательностью действий

Для отображения основных форм управления, таких как композиция, ветвление и повторение, в большинстве языков существует специальный набор управляющих операторов. Одним из основных аспектов каждого описываемого ниже оператора является то, что они представляют собой операторы с одной точкой входа и одной точкой выхода. Если один из таких операторов помещается в некоторую последовательность других операторов, тогда последовательность выполнения обязательно пройдет от предыдущего оператора через наш оператор с одной точкой входа и одной точкой выхода к следующему за ним оператору (при условии, что наш оператор не сможет включить в себя внутренний оператор goto, передающий управление в какое-либо иное место программы, отличное от единственной точки выхода из него). При чтении программы, составленной только из управляющих операторов с одной точкой входа и одной точкой выхода, порядок выполнения операторов программы должен соответствовать порядку операторов в тексте программы. Каждый оператор с одной точкой входа и одной точкой выхода может содержать внутреннее ветвление и внутренние циклы, но управление от одного такого оператора может переходить к другому только через его единственную точку выхода.

Старые языки, такие как COBOL и FORTARN, хотя и содержат несколько операторов с одной точкой входа и одной точкой выхода, но они все же сильно зависят от оператора goto и меток операторов. Оба эти языка было трудно адаптировать к современным языковым концепциям. Общее представление о языке COBOL можно получить из листинга 8.1 и обзора языка 8.3.

Обзор языка 8.3. COBOL

Возможности. В начале 60-х гг. язык COBOL (COmmon Business Oriented Language) широко использовался для обработки деловой информации на компьютерах (теперь это называют бизнес-приложениями).

История. Архитектура языка COBOL неоднократно пересматривалась — его первая версия вышла в 1960 г., а более поздние версии — в 1974 и 1984 гг. Разработка языка COBOL под руководством Грейса Хопера (Grace Hopper) была организована Министерством обороны США. Разработчики языка COBOL позаимствовали некоторые идеи из языка FLOWMATIC, созданного в компании Univac, включая использование существительных и глаголов для описания действий и отделение описаний данных от команд. При разработке языка COBOL была поставлена уникальная цель — создать язык программирования, использующий «естественный английский» для описания алгоритмов. Хотя получившийся язык достаточно удобен для чтения, но у него все же есть формальный синтаксис и программирование на нем требует определенных практических навыков.

Из-за большого количества представлений разнообразных данных и огромного числа вариантов для большинства операторов языка трансляция COBOL-программы в эффективный выполняемый код достаточно сложна. Большинство ранних компиляторов COBOL были крайне медленными, но более поздние усовершенствования методов компиляции привели к появлению относительно быстрых компиляторов языка COBOL, создающих довольно эффективный исполняемый код.

Пример. Программы COBOL организованы в виде четырех *разделов*. Такая организация отвечает двум основным целям разработки языка: отделение *машинно-зависимых* элементов программы от *машинно-независимых*, а также отделение описания данных от описания алгоритма. В результате появились три раздела программы: процедурный раздел (PROCEDURE division) содержит алгоритмы, раздел данных (DATA division) содержит описания данных, раздел окружения (ENVIRONMENT division) содержит машинно-зависимые программные спецификации, такие как связи между программой и внешними файлами данных. Четвертый раздел идентификации (IDENTIFICATION division) содержит название программы и имя ее автора, а также дополнительную информацию и документацию.

Строение языка COBOL основано на статической структуре времени выполнения. Не требуется организации управления ресурсами памяти во время выполнения программы, и многие аспекты языка были разработаны для того, чтобы позволить использовать относительно эффективные структуры времени выполнения (хотя эти цели не так важны, как аппаратная независимость и возможность переносимости программ).

Большинство программ читается легко, поскольку в этом языке используется синтаксис, подобный синтаксису английского языка. Для улучшения читаемости программы можно использовать многочисленные необязательные, так называемые *шумовые*, *слова*. Синтаксис языка COBOL обеспечивает легкую читаемость программы, однако затрудняет написание, потому что даже самая простая программа получается довольно длинной. В листинге 8.1 представлен краткий обзор синтаксиса языка COBOL.

Ссылка. J. E. Sammet "The early history of COBOL", *ACM History of Programming Languages Conference*, Los Angeles, CA (June 1978) *(SIGPLAN Notices(13)8 [august 1978])*, 121–161.

Листинг 8.1. Пример на языке COBOL

```
 1 IDENTIFICATION DIVISION.
 2 PROGRAM-ID. SUM-OF-PRICES.
 3 AUTHOR. T-PRATT.
 4 ENVIRONMENT DIVISION.
 5 CONFIGURATION SECTION.
 6 SOURCE-COMPUTER. SUN.
 7 OBJECT-COMPUTER. SUN.
 8 INPUT-OUTPUT SECTION.
 9 FILE-CONTROL.
10     SELECT INP-DATA ASSIGN TO INPUT.
11     SELECT RESULT-FILE ASSIGN TO OUTPUT.
12 DATA DIVISION.
13 FILE SECTION.
14 FD INP-DATA LABEL RECORD IS OMITTED.
15 01 ITEM-PRICE.
16     02 ITEM PICTURE X(30)
17     02 PRICE PICTURE 9999V99
18 WORKING-STORAGE SECTION.
19 77 TOT PICTURE 9999V99. VALUE 0. USAGE IS COMPUTATIONAL.
20 01 SUM-LINE.
21     02 FILLER VALUE ' SUM ='PICTURE X(12).
22     02 SUM-OUT PICTURE $$,$$$,$$9.99.
23     02 COUNT-OUT PICTURE ZZZ9.
24     ... Другие данные.
25 PROCEDURE DIVISION.
26 START.
27     OPEN INPUT INP-DATA AND OUTPUT RESULT-FILE.
28 READ-DATA.
29     READ INP-DATA AT END GO TO PRINT-LINE.
30     ADD PRICE TO TOT.
31     ADD 1 TO COUNT.
32     MOVE PRICE TO PRICE-OUT.
33     MOVE ITEM TO ITEM-OUT.
34     WRITE RESULT-LINE FROM ITEM-LINE.
35     GO TO READ-DATA.
36 PRINT-LINE.
37     MOVE TOT TO SUM-OUT.
38     ... Другие операторы.
39     CLOSE INP-DATA AND RESULT-FILE.
40     STOP RUN.
```

Составные операторы

Составной оператор представляет собой последовательность операторов, которую при конструировании более сложных операторов можно рассматривать как единый оператор. Обычно составные операторы записывают в виде

```
begin
...      – последовательность операторов (один или более)
end
```

В языках C, C++, Perl и Java составные операторы просто заключаются в фигурные скобки: {...}.

Операторы, входящие в составной оператор, записываются в той последовательности, в которой они должны выполняться. Таким образом, составной оператор является основной структурой для представления *композиции* операторов.

Поскольку составной оператор — это оператор, то группу операторов, представляющих собой логически завершенную часть вычисления, можно объединить вместе, используя для этого конструкцию begin ... end, причем можно составлять иерархические конструкции из таких групп.

Составные операторы реализуются на традиционном компьютере путем последовательного помещения в память блоков выполняемого кода, представляющих каждый из операторов составного оператора. Порядок, в котором они появляются в памяти, определяет порядок их выполнения.

Условные операторы

Условный оператор обеспечивает возможность альтернативного выполнения одного из двух или более операторов (ветвление) или выполнение какого-либо одного оператора при определенных условиях. В данном случае слово *оператор* обозначает либо *одиночный* элементарный оператор, либо составной оператор, либо другой управляющий оператор. Выбор осуществляется посредством проверки некоторого условия, которое обычно записывается в виде выражения, включающего в себя операцию отношения и булеву операцию. Наиболее часто в качестве условных операторов используются операторы if или case.

Операторы if. Необязательное выполнение одного оператора выражается оператором if с *одной ветвью*, а именно:

```
if выражение-условие then оператор endif
```

Выбор же из двух возможностей осуществляется при помощи оператора if с двумя ветвями:

```
if выражение-условие then оператор1 else оператор2 endif
```

В первом случае, если выражение-условие вычисляется равным *истина*, оператор выполняется, тогда как вычисление его равным значению *ложь* приводит к пропуску и невыполнению оператора из блока then. Во втором случае, в зависимости от вычисленного значения выражения-условия — *истина* или *ложь*, выполняется соответственно либо оператор$_1$, либо оператор$_2$.

Для выбора из нескольких вариантов необходимо в условный оператор if дополнительно вставить еще один оператор if или использовать оператор if из нескольких ветвей:

```
if условие1 then оператор1
      elsif условие2 then оператор2
      ...
      elsif условиеn then операторn
      else операторn+1 endif
```

Операторы case. Условия в операторе if, состоящем из нескольких ветвей, часто принимают вид многократной проверки значения одной и той же переменной, как, например, в следующем случае:

```
if Tag = 0 then оператор0
      elsif Tag = 1 then оператор1
      elsif Tag = 2 then оператор2
      else оператор3
      endif
```

Более лаконично такую структуру можно представить в виде оператора case, как это делается в языке Ada:

```
case Tag is
   when 0 => begin
      оператор0
      end;
   when 1 => begin
      оператор1
      end;
   when 2 => begin
      оператор2
      end;
   when others => begin
      оператор3
      end;
end case
```

Вообще, вместо переменной Tag может быть использовано любое выражение, значение которого вычисляется. Действия для каждого из возможных значений представляются сложным оператором, перед которым указывается соответствующее значение выражения. В качестве возможных значений, которые может принимать выражение в операторе case, очень удобно использовать перечисляемые типы или диапазоны целых значений. Например, если переменная Tag будет определена как целая переменная, принимающая значения из диапазона 0...5, то во время выполнения оператора case значения 0, 1 или 2 переменной Tag приведут соответственно к выполнению операторов $оператор_0$, $оператор_1$ или $оператор_2$, а все другие значения — к выполнению оператора $оператор_3$.

Реализация. Оператор if легко реализуется с помощью обычных команд перехода и ветвления, поддерживаемых аппаратной частью компьютера (аппаратная форма условного и безусловного goto). Оператор case во избежание повторной проверки значения одной и той же переменной обычно реализуется с помощью таблицы переходов. *Таблица переходов* — это вектор, компоненты которого размещены в памяти последовательно и каждый из них представляет собой безусловную команду перехода. После вычисления выражения, которое составляет условие оператора case, результат преобразуется в короткое целое число, представляющее собой сдвиг относительно базового адреса таблицы переходов. В результате выполнения команды перехода, хранящейся в области памяти таблицы, характеризуемой данным сдвигом, выполняется блок кода, который необходимо выполнить в случае выбора данного варианта. Окончательная структура реализации оператора case приведена на рис. 8.6.

Операторы цикла

В большинстве программ циклы служат основным механизмом для повторения вычислений. (Другой механизм — рекурсивные подпрограммы — описан в следующей главе.) Основная структура оператора цикла состоит из двух частей — *заголовка* и *тела*. Заголовок управляет количеством повторений тела, в то время как тело обычно является оператором (составным), который определяет действие оператора цикла. Хотя тело любого оператора цикла может быть произвольным, для заголовка, как правило, имеется всего несколько вариантов структуры.

Простое повторение. Простейший тип заголовка оператора цикла определяет, сколько раз должно выполняться тело. Оператор PERFORM языка COBOL является типичным примером этой конструкции:

```
perform body K times
```

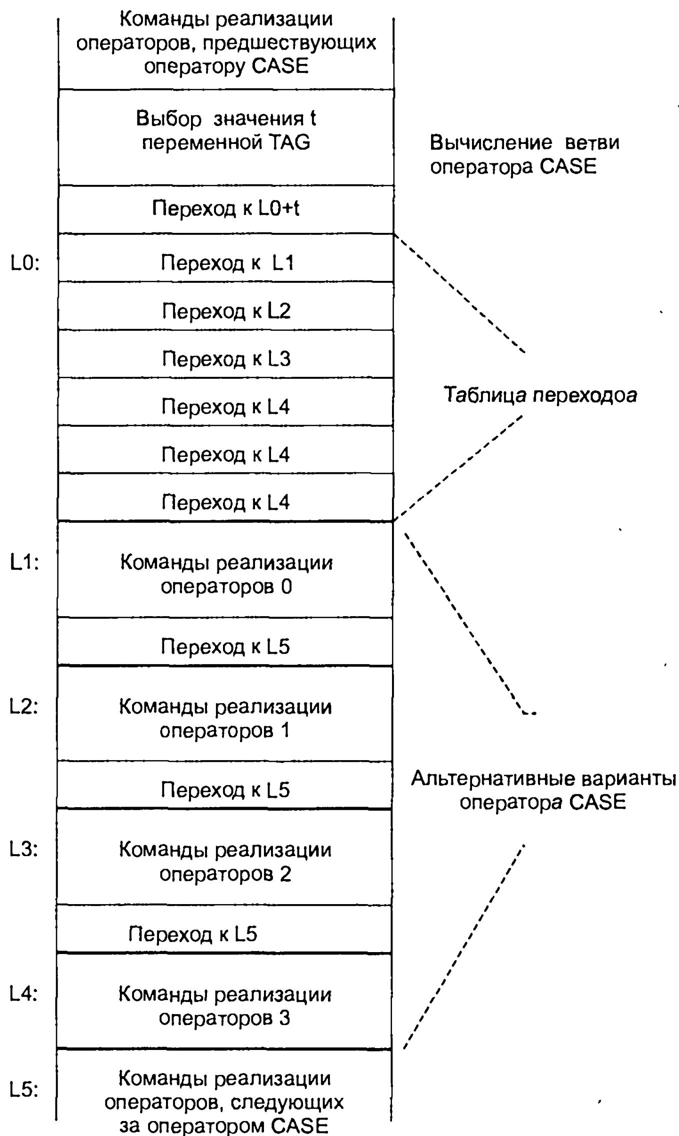


Рис. 8.6. Реализация таблицы переходов оператора case

Семантика оператора предписывает вычислить K, а затем выполнить тело оператора цикла K раз.

Однако даже в таком простом случае возникает несколько тонких вопросов. Можно ли значение K переопределить внутри тела и, таким образом, изменить количество повторений цикла? Что будет, если K равняется 0 или является отрицательным числом? Как это повлияет на процесс выполнения программы?

Хотя в случае такого простого оператора цикла эти вопросы могут показаться излишними, те же вопросы возникают и при использовании любой другой формы

оператора цикла, и поэтому важно их рассмотреть именно здесь, на самом простом примере. В каждом случае важно выяснить:

1. Когда происходит проверка на завершение цикла?
2. Когда вычисляются переменные, используемые в заголовке оператора?

Цикл с предусловием (while). Более сложные циклы можно организовать, используя в операторах цикла заголовок повторять пока. Типичный вид такого заголовка:

```
while условие do тело
```

В этом типе оператора цикла выражение, являющееся условием, вычисляется заново каждый раз после выполнения тела цикла. Обратите внимание на то, что в теле должны происходить какие-либо изменения значений некоторых переменных, используемых в выражении условия; иначе, однажды начавшись, этот цикл никогда не закончится.

Цикл с параметром (for). Третьей альтернативной формой оператора цикла является оператор, в заголовке которого указана переменная, которая используется в качестве счетчика (или индекса) во время вычислений. В таком заголовке задаются начальное значение счетчика, конечное значение и приращение, а тело цикла выполняется многократно. Сначала индекс принимает начальное значение, при следующем повторении выполнения тела цикла — начальное значение плюс приращение, затем — начальное значение плюс два приращения и т. д., пока значение счетчика не достигнет заданного конечного значения. В языке FORTRAN 77 это единственная форма оператора цикла. Типичную структуру такого оператора цикла можно проиллюстрировать на примере оператора for языка ALGOL:

```
for I := 1 step 2 until 30 do тело_цикла
```

Вообще говоря, начальное и конечное значения и приращение могут задаваться в виде произвольных выражений:

```
for K := N-1 step 2×(W-1) until M×N do тело_цикла
```

Здесь опять возникают вопросы о том, когда происходит проверка на завершение цикла и когда и как часто вычисляются различные выражения. Эти вопросы являются вопросами первостепенной важности и для разработчика языка, поскольку такие операторы цикла являются первыми кандидатами на оптимизацию, а ответы могут сильно повлиять на вид оптимизации, которую можно будет осуществить.

Повторение, основанное на данных. Иногда формат данных определяет счетчик повторений оператора цикла. Например, в языке Perl существует оператор foreach:

```
foreach $X (@arrayitem) { … }
```

При каждом повторении цикла скалярная переменная $X будет принимать значение, равное следующему элементу массива @arrayitem. Размер массива определяет количество повторений тела цикла.

Бесконечное повторение. Когда условия выхода из цикла сложны и не могут быть легко выражены в рассмотренных нами заголовках оператора цикла, обычно используется цикл с заголовком без явных критериев останова. Например, в языке Ada такой цикл имеет вид

```
loop
```

```
    exit when условие;
    ...
end loop;
```

В языке Pascal такой цикл организуется с помощью оператора while с условием, которое всегда выполняется:

```
while true do begin ... end
```

В языке С оператор for позволяет реализовать все рассмотренные концепции цикла в одной конструкции:

for(выражение$_1$; выражение$_2$; выражение$_3$){тело цикла}

Здесь выражение$_1$ является начальным значением, выражение$_2$ — условием повторения, выражение$_3$ — приращением. В языке С каждое из этих выражений не является обязательным, что обеспечивает гибкость построения циклов. Приведем несколько примеров циклов, написанных на языке С:

Простой счетчик от 1 до 10	`for(i=1; i<=10; i++){тело}`
Бесконечный цикл	`for(;;){тело}`
Счетчик с условием для выхода	`for(i=1; i<=100 && NotEndfile; i++){тело}`

Реализация операторов цикла. Реализация операторов цикла может быть проведена непосредственно с помощью аппаратных команд ветвления и перехода. Для реализации цикла for выражения из заголовка, определяющие конечное значение и приращение, должны быть вычислены при начальном входе в цикл и сохранены в специальной временной области памяти, откуда они извлекаются в начале каждой итерации для проверки значения переменной-счетчика и добавления приращения к ней.

Проблемы структурного управления последовательностью действий

Оператор goto обычно рассматривают как последнюю возможность, когда описанные выше структурные операторы управления не позволяют адекватно выразить сложную структуру управления последовательностью действий. Хотя теоретически любую такую структуру всегда можно выразить, используя только структурные формы операторов, на практике сложная форма может не иметь прямого и естественного представления с помощью одних лишь подобных операторов. Несколько таких проблемных областей известны, и часто для этих случаев предусматриваются специальные управляющие конструкции, которые полностью исключают использование оператора goto. Приведем некоторые наиболее общие случаи.

Циклы с множественным выходом. Часто завершение выполнения цикла может происходить при выполнении нескольких независимых условий. Распространенный пример — цикл поиска: в векторе из K элементов осуществляется поиск первого элемента, удовлетворяющего некоторому условию. Цикл заканчивается либо если вектор проверен до конца, либо если найден соответствующий элемент. Перебор элементов вектора естественным образом выражается циклом for:

```
for I := 1 to K do
    if VECT[I] = 0 then goto α
```

где α — некоторая метка оператора вне тела цикла.

Однако в языке Pascal либо оператор goto следует использовать для выхода из оператора цикла до его завершения, либо цикл for нужно заменить циклом while, который скрывает информацию о наличии индексной переменной I и диапазоне ее значений, содержащуюся в заголовке цикла for.

Оператор exit языка Ada и оператор break языка С представляют альтернативную конструкцию для выражения способа такого преждевременного выхода из цикла без использования оператора goto:

```
for I in 1..K loop
   exit when VECT(I) = 0
end loop;
```

Конструкция do-while-do. Иногда проверять условие выхода из цикла удобно не в начале и не в конце цикла, а где-нибудь в середине, после того как выполнена некоторая обработка данных, например:

```
loop
   read(X)
   if end_of_file then goto α {α – вне цикла}
   process(X)
end loop;
```

Эта форма иногда называется do-while-do, потому что возможность выхода может определяться оператором while, вставленным в середину цикла:

```
dowhiledo
   read(X)
   while (not end_of_file)
   process(X)
end dowhiledo
```

К сожалению, ни один из стандартных языков не реализует эту структуру, хотя конструкция языка С if (условие) break и оператор языка Ada exit when по смыслу близки к ней.

Исключительные ситуации. Исключения (или исключительные ситуации) могут представлять собой различные ситуации, приводящие к возникновению ошибок, например, неожиданное достижение конца файла, ошибки, связанные с выходом индексов из диапазонов их значений, или обработка некорректных данных. Операторы, обрабатывающие такие исключительные ситуации, часто группируются в определенном месте программы (например, в конце подпрограммы), в которой может возникнуть ошибка, а иногда даже в отдельную подпрограмму, занимающуюся только обработкой исключительных ситуаций. В языках, в которых не реализованы специальные операторы обработки исключительных ситуаций, лучше всего с помощью оператора goto передавать управление от места, где возникла исключительная ситуация, обработчику исключений (группа операторов). Однако в языках Ada и ML существуют специальные языковые механизмы для определения обработчика исключений и передачи управления при обнаружении исключения. Типичным оператором в алгоритме обработки исключений в языке Ada является оператор raise:

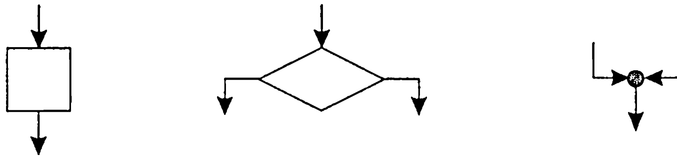
```
raise BAD_CHAR_VALUE
```

Этот оператор передает управление операторам обработки исключительной ситуации, соответствующей исключению с именем BAD_CHAR_VALUE. Исключения и их обработка будут рассмотрены далее в разделе 11.1.1.

8.3.3. Первичные программы

Хотя на первый взгляд совокупность управляющих структур, представленная в этой главе, кажется некоторым случайным набором операторов, для описания непротиворечивой теории структур управления можно использовать теорию *первичных, или атомарных, программ.* Теория первичных программ была предложена Маддуксом (Maddux) [76] в качестве обобщения методологии структурного программирования для определения однозначной иерархической декомпозиции блок-схем.

Мы предполагаем, что графы программ могут содержать три класса узлов:



Функциональные узлы представляют вычисления, производимые программой, и изображаются прямоугольниками с одной входящей в этот узел дугой и одной выходящей. Интуитивно понятно, что функциональные узлы представляют операторы присваивания, выполнение которых вызывает изменение состояния виртуальной машины.

Узлы принятия решения изображаются в виде ромбов с одной входящей дугой и двумя выходящими, помечаемыми *истина* и *ложь*. Эти узлы представляют предикаты, и управление из узла принятия решения передается дальше либо по ветви *истина*, либо по ветви *ложь*.

Узел соединения представляется в виде точки, в которой сходятся две дуги графа, чтобы сформировать одну выходную дугу.

Любая блок-схема состоит их этих трех компонентов. Определим *правильную программу*, являющуюся формальной моделью структуры управления, как блок-схему, которая:

1) имеет одну входящую дугу;
2) имеет одну выходящую дугу;
3) имеет путь от входящей дуги к любому узлу и из любого узла — к выходящей дуге.

Нашей целью является определить различие между структурированной правильной программой и неструктурированной. Например, если посмотреть на рис. 8.7, то можно заметить очевидные различия между блок-схемами на рис. 8.7, *а* и 8.7, *б*. Левая блок-схема содержит большое количество казалось бы случайным образом построенных дуг графа, в то время как правый граф имеет упорядоченную вложенную структуру. С помощью первичных программ можно объяснить эти различия.

Первичная программа является правильной программой, которую нельзя подразделить на более мелкие правильные программы. Если нельзя разрезать две дуги правильной программы таким образом, чтобы разделить ее на отдельные графы, то правильная программа является первичной. (Единственным исключением из этого правила является длинная последовательность функциональных

узлов, которая считается одной первичной программой.) Программа, изображенная на рис. 8.7, *а*, является первичной, а на рис. 8.7, *б* — нет. Пунктирные прямоугольники A, B и C представляют вложенные правильные программы внутри всей блок-схемы.

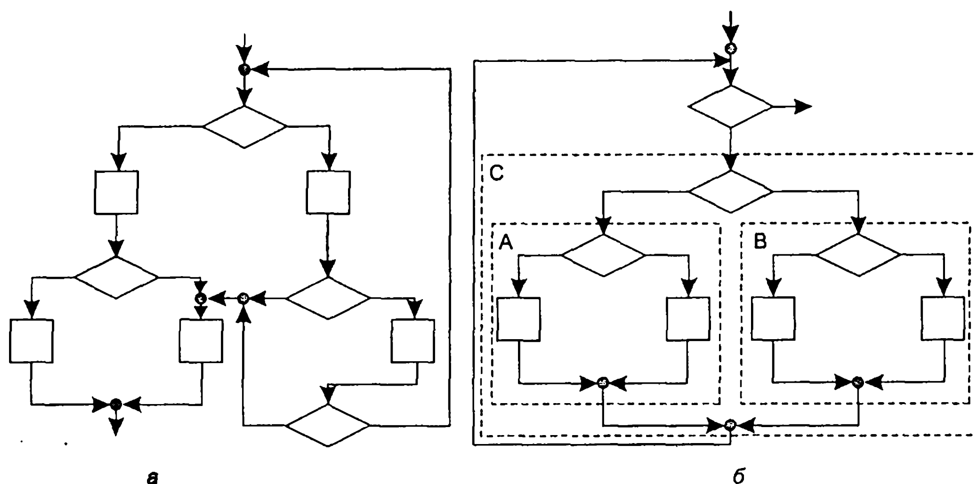


Рис. 8.7. Блок-схемы: *а* — первичная программа; *б* — составная программа

Определим *составную программу* как правильную программу, которая не является первичной. На рис. 8.7, *б* изображена как раз составная программа. Заменив каждый первичный компонент правильной программы функциональным узлом (например, первичные компоненты A и B на рис. 8.7, *б* заменим функциональными узлами), мы можем продолжить этот процесс (например, заменить теперь блок C как первичный компонент функциональным узлом) до тех пор, пока не получим представление любой правильной программы в виде единственной первичной программы.

Можно перечислить все первичные программы. На рис. 8.8 изображены все первичные программы, которые включают в себя не более четырех узлов. Обратите внимание на то, что большинство из них либо неэффективны, либо являются стандартными структурами управления, описанными ранее в этой главе. Первичные программы *а*, *б*, *д* и *к* представляют собой последовательности функциональных узлов и соответствуют основному блоку большинства языков программирования. Первичная программа *е* — конструкция `if-then`, *ж* — `do-while`, *з* — `repeat-until`, *л* — `if-then-else`, *м* — `do-while-do`.

Рассмотрим первичные программы *в*, *г* и с *н* по *т*. Все они состоят только из узлов принятия решения и соединения. В них нет функциональных узлов, значит, они не изменяют пространство состояний виртуальной машины. Поскольку эти первичные программы не изменяют значений никаких данных, все они вычисляют тождественную функцию. Однако из *в* и *н* всегда существует выход (то есть выполнение завершается), и, следовательно, они являются тождественными функциями при любых входных данных, тогда как остальные первичные программы могут выполняться циклически. Однажды начавшись, эти циклы будут выполнять-

ся и никогда не завершатся. Они представляют собой частично вычислимые функции, которые завершаются только при определенных значениях входных данных. Ни один из этих вариантов первичных программ не представляет эффективной структуры управления в программе.

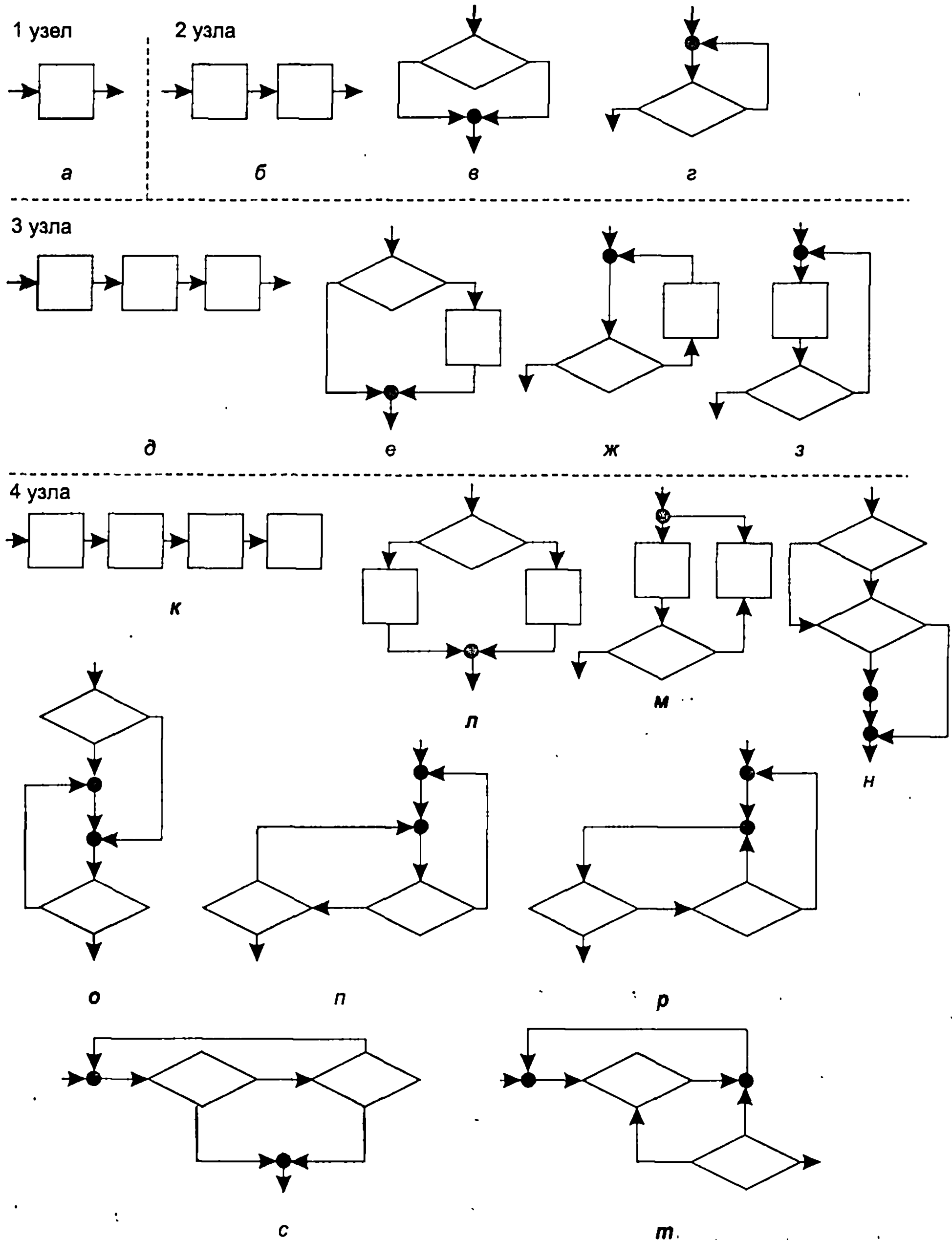


Рис. 8.8. Виды первичных программ

Не удивительно, что в определения языков программирования были включены наборы управляющих структур, описанные в этой главе. Все они представляют собой первичные программы с небольшим количеством узлов. Такие первичные программы просты для понимания, и те изменения в пространстве состояний виртуальной машины, которые являются результатом их выполнения, поддаются управлению. Перечислив все первичные программы, мы приходим к очевидному выводу, что конструкция do-while-do является естественной структурой управления, хотя, к сожалению, она была проигнорирована разработчиками языков.

Структурная теорема. Когда в 70-е гг. разрабатывалась методология структурного программирования, обсуждался вопрос, не приведет ли использование только таких структур управления к ограничению возможностей программ. То есть можно ли написать программу, подобную приведенной на рис. 8.7, *а*, используя только структуры управления, представленные на рис. 8.8? На этот вопрос отвечает теорема Бема–Якобини (Bohm–Jacobini theorem) [19]. Они показали, что любую первичную программу можно преобразовать в другую первичную программу, используя только операторы while и if. На рис. 8.9 приведен набросок процесса, похожий на конструкцию Бема–Якобини (Bohm–Jacobini), однако это доказательство принадлежит Харлану Миллзу (Harlan Mills) [71].

1. В любой заданной блок-схеме пометьте все узлы. Выходящую дугу пометьте числом 0.
2. Определите новую переменную I.
3. Для каждого узла блок-схемы примените преобразование, приведенное на рис. 8.9, *а*.
4. Перестройте программу, как показано на рис. 8.9, *б*.

Должно быть ясно, что переменная I играет роль счетчика команд виртуальной машины, указывающего на следующий оператор, который необходимо выполнить, а вся программа является просто рядом вложенных в единственный цикл while операторов if. Эта программа будет работать так же, как и исходная блок-схема, с единственным отличием — переменная I будет изменяться в процессе ее выполнения.

Результаты исследований Бема и Якобини иногда приводят в качестве причины, почему *не нужно* избегать операторов goto. Можно написать программу как с использованием оператора goto, так и без него, а затем, пользуясь структурной теоремой, переделать программу в «хорошо структурированную». Доверчивым людям было продано много таких структурирующих машин. Структурное программирование не является синонимом хорошего программирования. Оно обозначает только использование структур управления, которые являются первичными программами с небольшим количеством узлов. Если начинать с плохого кода «спагетти», то преобразование просто приведет к плохому структурированному коду.

Терема Бема–Якобини фактически доказывает лишь то, что любую программу *можно* написать, используя только стандартные структуры управления. Приведенный алгоритм не обеспечивает наилучшее решение. Его разработка полностью за-

висит от программиста. Мы знаем лишь то, что существует некоторое решение *не хуже*, чем результат Бема и Якобини, а может быть, даже лучше.

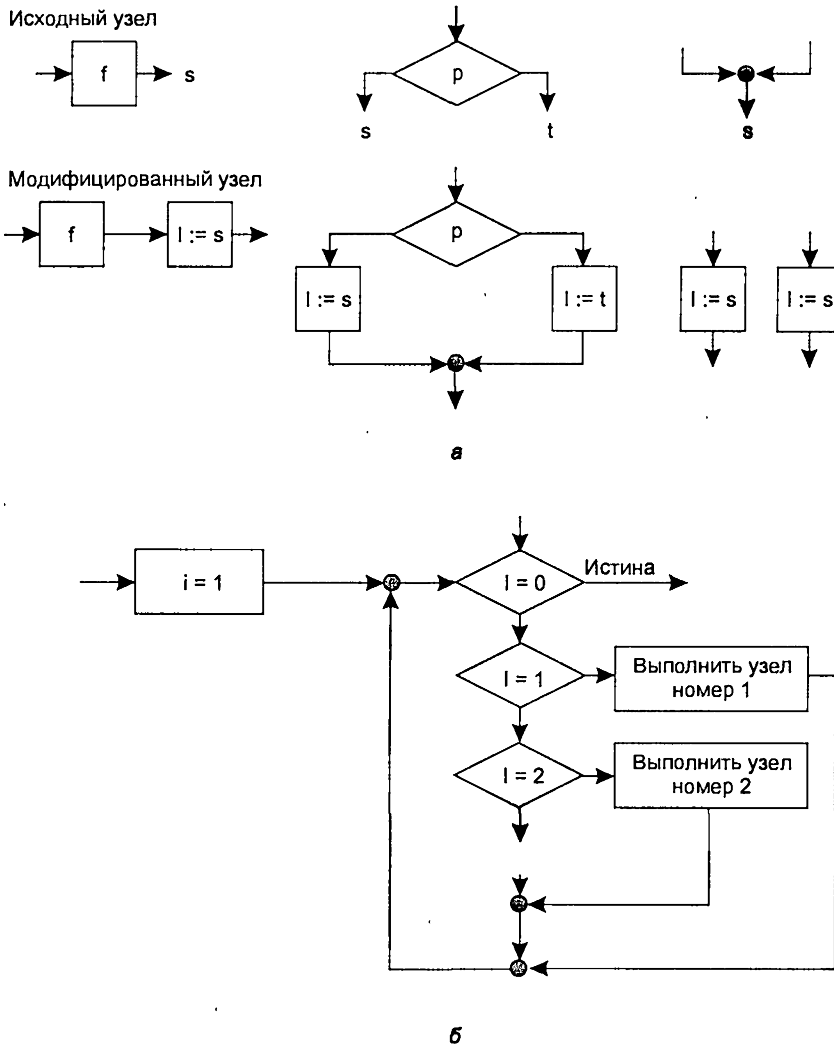


Рис. 8.9. Структурная теорема: *а* — преобразования; *б* — модифицированная программа

8.4. Последовательность вычисления неарифметических выражений

В предыдущем разделе обсуждалась последовательность выполнения действий при вычислении арифметических выражений. Однако в языках присутствуют и другие форматы выражений. В такие языки, как Prolog и ML, разработанные для обработки символьных данных, включены другие формы вычисления выражений.

8.4.1. Обзор языка Prolog

В отличие от других языков, описанных в этой книге, Prolog не является универсальным языком программирования, но зато он ориентирован на решение задач с использованием исчисления предикатов.

Целью разработки языка Prolog было предоставить возможность задания спецификаций решения и позволить компьютеру вывести из них последовательность выполнения для этого решения, а не задание алгоритма решения задачи, как в большинстве изученных нами языков. Например, если информация об авиарейсах представлена в следующей форме:

```
flight(номер_рейса. пункт_отправления. пункт_назначения. время_отправления.
   время_прибытия)
```

тогда все рейсы из Лос-Анджелеса в Балтимор можно задать либо в виде прямых рейсов через оператор

```
flight(номер_рейса. Лос-Анджелес. Балтимор. время_отправления. время_прибытия)
```

либо в виде рейсов с промежуточной посадкой

```
flight (номер_рейса1. Лос-Анджелес. X. время_отправления1. время_прибытия1).
flight (номер_рейса2. X. Балтимор. время_отправления2. время_прибытия2).
время_отправления2 >= время_прибытия1+30
```

Это означает, что вы определяете город X, в который можно попасть рейсом из Лос-Анджелеса и откуда можно улететь в Балтимор, причем самолет в Балтимор вылетает по крайней мере через 30 минут после прилета рейса из Лос-Анджелеса, чтобы осталось время на пересадку. Здесь не задан никакой алгоритм, заданы только условия для получения правильного решения. Если мы сможем задать подобный набор условий, язык сформирует последовательность действий, необходимую для выбора подходящего рейса.

История. Разработка языка Prolog началась в 1970 г. Аланом Кулмероэ и Филиппом Русселом (Alan Coulmerauer and Philippe Roussel). Они хотели создать язык, который мог бы делать логические заключения на основе заданного текста. Название Prolog является сокращением от «PROgramming in LOGic». Этот язык был разработан в Марселе (Франция) в 1972 г. Принцип резолюции Ковальского (Kowalski), сотрудника Эдинбургского университета (раздел 8.4.5), казался подходящей моделью, на основе которой можно было разработать механизм логических выводов. С ограничением резолюции на дизъюнкты Хорна унификация (раздел 8.4.3) привела к эффективной системе, где неустранимый недетерминизм обрабатывался с помощью процесса отката, который мог быть легко реализован. Алгоритм резолюции позволял создать выполняемую последовательность, необходимую для реализации спецификаций, подобных приведенному выше отношению flight.

Первая реализация языка Prolog с использованием компилятора Вирта (Wirth) ALGOL-W была закончена в 1972 г., а основы современного языка были заложены в 1973 г. Использование языка Prolog постепенно распространялось среди тех, кто занимался логическим программированием, в основном благодаря личным контактам, а не через коммерциализацию продукта. В настоящее время существует несколько различных, но довольно похожих между собой версий. Хотя стандарта языка Prolog не существует, однако версия, разработанная в Эдинбургском университете, стала наиболее широко используемым вариантом. Недостаток разработок эффективных приложений Prolog сдерживал его распространение вплоть до 1980 г.

Краткий обзор языка. Программа на языке Prolog состоит из набора фактов, определенных отношений между объектами данных (фактами) и набором правил (образцами отношений между объектами базы данных). Эти факты и правила вводятся в базу данных через операцию `consult`. Для работы программы пользователь должен ввести запрос — набор термов, которые все должны быть истинны. Факты и правила из базы данных используются для определения того, какие подстановки для переменных в запросе (называемые *унификацией*) согласуются с информацией в базе данных.

Язык Prolog, как интерпретатор, приглашает пользователя вводить информацию. Пользователь набирает запрос или имя функции. Выводится значение (*истина* — `yes`, или *ложь* — `no`) этого запроса, а также возможные значения переменных запроса, присвоение которых делает запрос истинным (то есть *унифицирует* запрос). Если ввести символ «;», тогда отображается следующий набор значений переменных, унифицирующий запрос, и так до тех пор, пока не исчерпается весь набор возможных подстановок, после чего Prolog печатает `no` и ждет следующего запроса. Возврат каретки воспринимается как прекращение поиска дополнительных решений.

Хотя выполнение программы на языке Prolog основывается на спецификации предикатов, оно напоминает выполнение программ на аппликативных языках LISP или ML. Разработка правил языка Prolog требует того же рекурсивного мышления, что и разработка программ на этих аппликативных языках.

Язык Prolog имеет простые синтаксис и семантику. Поскольку он ищет отношения между некоторыми рядами объектов, основными структурами данных являются переменная и список. Правило ведет себя подобно процедуре, за исключением того, что концепция унификации более сложна, чем относительно простой процесс подстановки параметров в выражения (см. раздел 8.4.3).

8.4.2. Сопоставление с образцом

В таких языках, как Perl, Prolog и ML, ключевой операцией является операция сопоставления с образцом. В этом случае операция выдает положительный результат путем сопоставления и присваивания набора переменных заранее определенному шаблону. Образцом этой операции является распознавание деревьев синтаксического разбора в НФБ-грамматиках, описанных в разделе 3.3.1.

Например, следующая грамматика распознает палиндромы нечетной длины на алфавите, состоящем из 0 и 1:

$A \rightarrow 0A0 \mid 1A1 \mid 0 \mid 1$

Правильная строка 00100 распознается следующим образом:

A_1 сопоставляется с центром 1
A_2 сопоставляется с $0A_10$
A_3 сопоставляется с $0A_20$

Из этих трех присваиваний A_1 значения 1, A_2 значения $0A_10$ и A_3 значения $0A_20$ мы можем сконструировать полное дерево синтаксического разбора этой строки (рис. 8.10). Мы уже обсуждали операцию сравнения с образцом с помощью регулярных выражений Perl в разделе 3.3.3.

SNOBOL4 — это язык, разработанный для прямого моделирования этой возможности, как показано в обзоре языка 8.4. Обратите внимание на то, что приве-

денная программа состоит всего из 9 операторов, и вряд ли на любом из существующих языков можно было бы написать программу, решающую ту же самую задачу и состоящую из такого небольшого и в то же время мощного набора операторов.

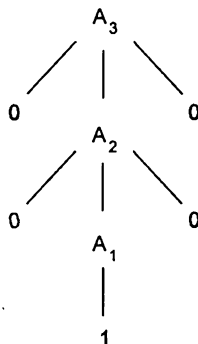


Рис. 8.10. Сопоставление с образцом в языке SNOBOL4

Также SNOBOL4 обладает еще одним интересным свойством. Его реализация не зависит от конкретной машинной архитектуры. Она разработана для виртуальной машины обработки строк (как показано на рис. 2.4 в разделе 2.2.2). Все, что требуется для запуска SNOBOL4, — это реализовать операции обработки строк виртуальной машины в виде макросов имеющегося компьютера. Благодаря этому SNOBOL4 был одним из первых языков, который:

1) был доступен на любом компьютере;
2) в любой реализации имел абсолютно одинаковую семантику.

Если в языке SNOBOL4 для выполнения операции сравнения с образцом используется замещение строк, то в языке Prolog в качестве механизма сопоставления используется понятие отношения как множества n-кортежей. Из известных экземпляров этих отношений (называемых *фактами*) можно вывести другие экземпляры. Например, можно рассмотреть отношение `ParentOf` (Родитель), для которого известны следующие факты:

`ParentOf(John, Mary)`	Джон — родитель Мери
`ParentOf(Susan, Mary)`	Сюзанна — родитель Мери
`ParentOf(Bill, John)`	Билл — родитель Джона
`ParentOf(Ann, John)`	Анна — родитель Джона

Для нахождения родителя Мери нужно просто написать отношение `ParentOf(X, Mary)`, а Prolog попытается определить значение `X` из известного набора фактов, находящихся в базе данных, и сделает вывод, что `X` может быть либо Джон, либо Сюзанна. Если мы хотим узнать обоих родителей Мери, то нужен более сильный оператор. Мы можем разрабатывать предикаты, включающие несколько фактов из базы данных. Итак, если мы хотим узнать двух различных родителей Мери, то нужно записать:

```
ParentOf(X, Mary), ParentOf(Y, Mary), not(X = Y).
```

Здесь запятая, разделяющая предикаты, является логической операцией И (то есть для истинности всего отношения все предикаты должны быть истинными).

Обзор языка 8.4. SNOBOL4

Возможности. Сопоставление с образцом на основе НФБ-грамматик. Полностью динамический язык, включая объявления, типы, распределение памяти, даже точки входа и выхода из процедуры. Реализация использует виртуальные макрокоманды обработки строк — простой перезаписью макрокоманд для любого существующего компьютера.

История. Разработка началась в 1962 г. Ральфом Грисвольдом (Ralf Griswold), Иваном Полонским (Ivan Polonsky) и Дэвидом Фарбером (David Farber), сотрудниками лаборатории AT&T Bell Labs. Их целью было создание языка обработки строк для работы с формулами и анализа графиков. В 1950 г. Ингве (Yngve) из MIT разработал язык COMIT для обработки естественных языков на основе правил НФБ, однако группа из Bell Labs сочла COMIT слишком ограниченным для своих целей.

Изначально язык назывался SCL7 (Symbolic Computation Language 7), затем его название сменилось на SEXI (String Expression Interpreter), которое по понятным причинам было осуждено в 60-е гг., и, наконец, он стал называться SNOBOL (StriNg Oriented symBOlic Language) — искусственно созданный акроним, лишенный интуитивно понятного смысла. Было разработано несколько версий языка SNOBOL — SNOBOL, SNOBOL2, SNOBOL3 и SNOBOL4. Последний пользовался успехом в 70-е гг.

Пример. Найти среди вводимых строк палиндром, составленный из 0 и 1, максимальной нечетной длины:

```
START     GRAMMAR = 0 | 1 | 0 *GRAMMAR 0 | 1 *GRAMMAR 1
*        Устанавливает в качестве образца НФБ-грамматику
LOOP      NEWLINE = TRIM(INPUT)        : F(END)
*        Получает следующую строку без завершающих пробелов.
*        В случае ошибки переход на END.
        NEWLINE (POS(0) SPAN("01") PROS(0))     : F(BAD)
*       Проверяет строку на наличие только нулей и единиц.
*       SPAN - строка из нулей и единиц.
*       POS(0) - первая позиция. PROS(0) - последняя.
        SN = SIZE(NEWLINE)
NEXT      NEWLINE POS(0) GRAMMAR . PALINDROME POS(SN)
-         :S(OK) F(NOTOK)
*       Строка проверяется на соответствие грамматике через POS(SN)
*       Если сравнение не прошло, переход в последнюю позицию.
*       Если успешно, печатается ответ.
*       Совпавшая часть присваивается PALINDROME
OK      OUTPUT="MATCH: " PALINDROME      :(LOOP)
NOTOK     SN = SN - 1                                 :(NEXT)
BAD     OUTPUT = "IMPROPER INPUT: " NEWLINE    :(LOOP)
END
```

Пример выполнения:

Ввод:	Вывод:
11011	MATCH: 11011
11011101	MATCH: 11011
11211	IMPROPER INPUT: 11211

Ссылка. R. E. Griswold "A history of the SNOBOL programming language", *ACM History of Programming Languages Conference*, Los Angeles, CA (June 1978) *(SIGPLAN Notices (13)8 [August 1978])*, 275–308.

Сила языка Prolog заключается в построении отношений, которые могут быть логически выведены из известного набора фактов. Эти отношения могут быть построены из других отношений, как, например, GrandparentOf (дед или бабка):

```
GrandparentOf(X, Y) :- ParentOf(X, Z), ParentOf(Z, Y).
```

Эта запись обозначает, что отношение GrandparentOf определяется (что обозначается символом :-) как два отношения ParentOf таких, что существует некоторый объект Z, являющийся родителем Y, но для него самого родителем является X. Таким образом, выполнение GrandparentOf(X,Mary) приведет к тому, что X будет либо Bill, либо Ann, а выполнение GrandparentOf(Y, John) приведет к ошибке, поскольку в нашей базе данных не содержится соответствующего факта.

Перезапись элементов

Перезапись элементов является ограниченной формой операции сопоставления с образцом, которая широко применяется в языках программирования. Пусть заданы строка $a_1a_2 \ldots a_n$ и правило перезаписи $\alpha \Rightarrow \beta$, если $\alpha = a_i$; тогда говорят, что строка $a_1 \ldots a_{i-1}\beta \ldots a_n$ представляет *перезапись элементов* строки $a_1a_2 \ldots a_n$. Сопоставление запроса языка Prolog с правилом в базе данных является формой перезаписи (с подстановкой; см. раздел 8.4.3).

Перезапись элементов уже рассматривалась нами в разделе 3.3.1 в связи с НФБ-грамматиками и синтаксическим разбором. Генерирование вывода цепочки в языке с заданной НФБ-грамматикой является просто формой процесса перезаписи. Например, если задана грамматика

```
A → 0B | 1
B → 0A
```

можно сгенерировать цепочку 001 посредством следующего вывода:

A ⇒ 0B	Используется правило A → 0B
0B ⇒ 00A	Используется правило B → 0A
00A ⇒ 001	Используется правило A → 1

В языке ML перезапись элементов используется как форма определения функции. Например, функция вычисления факториала в языке ML может быть определена довольно просто:

```
fun factorial(n:int) = if n=1 then 1 else n*factorial(n-1);
```

Здесь область определения функции factorial состоит из двух частей: при n = 1 возвращается значение 1, при всех остальных положительных целых возвращаемое значение задается выражением n*factorial(n - 1). Два варианта разделяются оператором if. Эту функцию можно рассматривать как две отдельные функции — константа 1 для множества {1} и значение n*factorial(n - 1) для всех остальных целых чисел (то есть {n | n > 1}).

Вместо того чтобы использовать оператор if для разделения области определения, можно использовать перезапись элементов для указания каждого отдельного случая в определении функции:

```
fun fact(1) =                 1
  | fact(N:int) =             N * fact(N - 1)
```

Здесь каждая подобласть определения отделяется символом |. Язык ML заменяет вызов функции на соответствующее определение.

В качестве другого примера рассмотрим функцию length языка ML, она является композицией перезаписи элементов и более общей операции сравнения с образцом:

```
fun length(nil) =            0
      | length(a::y) =       1+length(y);
```

nil обозначает пустой список, a :: — операцию конкатенации списка. То есть a :: [b,c,d] = [a,b,c,d]. В данном случае, если область определения функции length — пустой список, тогда значение будет равно 0. Иначе если область является списком, состоящим хотя бы из одного элемента (a), тогда значение функции будет равно 1 плюс длина остатка списка. Язык ML автоматически сопоставляет аргумент функции length и присваивает голову списка переменной a, а хвост — переменной y. Эта возможность становится очень важной, когда речь заходит о полиморфизме в языке ML (раздел 7.3).

8.4.3. Унификация

База данных языка Prolog состоит из *фактов*, таких как ParentOf(Ann, John), и *правил*, таких как

```
GrandparentOf(X, Y):- ParentOf(X, Z), ParentOf(Z, Y)
```

Выражение, содержащее одну или более переменных, например GrandparentOf (X, John), называется *запросом* и представляет собой неизвестное отношение. (Более точное описание такого запроса мы дадим в разделе 8.4.5 при обсуждении принципа резолюции, управляющего выполнением программы на языке Prolog.) Основной особенностью языка Prolog является использование операции сравнения с образцом для выяснения, разрешается ли запрос каким-либо фактом из базы данных или факт может быть выведен путем применения правил, содержащихся в базе данных, к другим фактам и правилам. В языке Prolog используется операция *унификации* — подстановки переменных в отношения — для определения (путем сравнения с образцом), имеется ли для данного запроса корректная подстановка, согласующаяся с правилами и фактами базы данных.

Рассмотрим приведенное выше отношение ParentOf. Допустим, мы хотим разрешить запрос:

```
ParentOf(X, Mary) = ParentOf(John, Y).
```

В данном случае отношение-факт ParentOf(John,, Mary) является решением для обеих частей запроса. Говорят, что *экземпляр* ParentOf(John, Mary) *унифицирует* ParentOf(X, Mary) и ParentOf(John, Y) подстановкой John вместо X, и Mary вместо Y, поскольку оба эти отношения после соответствующей подстановки выдают этот факт. Об унификации можно рассуждать как о расширении общего свойства подстановки.

Подстановка. Подстановка является одним из первых принципов, которые изучаются в программировании. Подстановка — это основной принцип, лежащий в основе передачи параметров и создания макрорасширений. Например, рассмотрим макрос языка C[1]:

```
#define mymacro(A,B,C) printf("Binary %d%d%d is %d\n", A, B, C, 4*(A)+2*(B)+(C))
```

[1] printf печатает выходную строку. Функция печатает свой первый строковый аргумент, используя встроенные коды для определения, в какое место выходной строки поместить значения последующих аргументов. Последовательность символов %d обозначает, что надо печатать следующий аргумент в списке как целый. Символы \n обозначают конец строки.

Данная строка означает, что когда в программе на языке C используется mymacro(..), первый фактический параметр mymacro заменяет параметр A, второй фактический параметр заменяет параметр B, третий — C. То есть запись mymacro(1. 0. 1) равносильна следующей:

```
printf("Binary %d%d %d is %d\n". 1. 0. 1. 4*(1)+2*(0)+(1))
```

Поскольку строковый параметр функции printf также является подстановкой, то предыдущая запись равносильна следующей:

```
printf("Binary 101 is 5\n")
```

Подставляемые параметры не обязательно должны быть целыми числами. *Любая* строка может быть подставлена в макрорасширение вместо A, B и C. Таким образом, макрос mymacro(X + Y.Z/2. Myvar + 3) эквивалентен следующей записи:

```
printf("Binary %d%d%d is %d\n". X+Y. Z/2. Myvar + 3. 4 * (X + Y) + 2 * (Z/2) + (Myvar + 3))
```

Важно помнить, что мы подставляем произвольное выражение вместо одной переменной в определении макроса.

Давайте расширим эту задачу до *двух* следующих макроподстановок:

```
mymacro(X + Y.Z/2. ?):
mymacro(?. ?. Myvar + 3):
```

Символы ? представляют неизвестные подстановки. Возможно ли, чтобы оба эти макрооператора представляли одинаковые вычисления? Это и есть основной принцип, лежащий в основе унификации. Если мы примем, что C представляет Myvar + 3 в первом макрорасширении, а A и B представляют X + Y и Z/2 соответственно — во втором, тогда оба макрооператора будут представлять одинаковые вычисления. Однако для

```
mymacro(X + Y.Z/2. ?):
mymacro(?. Z + 7. ?):
```

не существует набора присваиваний переменным A, B или C такого, чтобы эти макрооператоры представляли собой одинаковые вычисления. Суть унификации заключается в определении правильного набора подстановок вместо знаков вопроса. В то время как подстановка есть результат применения новых значений к параметрам макроса-образца, унификация есть результат одновременных подстановок в несколько макросов-образцов, имеющих целью показать, что все они эквивалентны при некотором наборе одновременных подстановок.

Общая унификация. Как отмечалось ранее, чтобы унифицировать два выражения U и V, нужно найти такие подстановки для переменных в этих выражениях, которые делают оба выражения одинаковыми. Например, для унификации f(X. John) и f(g(John). Z) свяжем X с g(John), а Z — с John и в качестве унифицированного экземпляра обоих выражений получим f(g(John). John). Если мы унифицируем выражения U и V, то зачастую обозначаем сделанные подстановки буквой σ (сигма) и записываем $U\sigma = V\sigma$.

Разница между подстановкой и унификацией демонстрируется на рис. 8.11. Когда мы применяем подстановку, у нас есть некоторое определение-шаблон (например, F(A. B)), которое может представлять собой сигнатуру подпрограммы или макроопределение, и экземпляр шаблона (например, F(g(i). h(j))), который может представлять вызов подпрограммы или расширение макроса. Подстановка требует переименования параметров определения-шаблона в действительные зна-

чения этого экземпляра. Однако в случае унификации обычно имеются два отдельных определения-шаблона (например, F(A, B) и M(C, D)) и экземпляр шаблона (например, F(John, M(h(v), 7))). Вопрос в этом случае ставится следующим образом: возможно ли присвоить какие-либо значения переменным A, B, C и D, так чтобы экземпляр шаблона стал подстановкой для обоих определений-шаблонов?

Образец	F(A, B) = G(A, B)	**Образец**	F(A, B) = G(A, B) M(C, D) = N(C, D)
Пример	F(g(i), h(j))	**Пример**	F(John, M(h(v), 7))
Подстановка	g(i) for A h(j) for B F(A, B) = G(g(i), h(j))	**Подстановка**	John for A M(h(v), 7) for B F(A, B) = G(John, M(h(v), 7)) h(v) for C 7 for D M(C, D) = N(h(v), 7)
		Унификация	F(John, N(h(v), 7)) (John — A; N(h(v), 7) — B; h(v) — C; 7 — D) G(John, M(h(v), 7)) (John — A; M(h(v), 7) — B; h(v) — C; 7 — D)

Рис. 8.11. Различие между подстановкой и унификацией

Для применения определения шаблонов, возможно, мы должны сделать подстановки в *обоих* направлениях. Так, в приведенном примере если мы в определение F подставим John вместо A, в определение M подставим 7 вместо D, а также в определение F подставим шаблон M(h(v), 7) вместо B и в определение M подставим h(v) вместо C, то наш экземпляр шаблона будет представлять правильную подстановку для обоих исходных определений-шаблонов F и M. Из нашего исходного выражения F(John, M(h(v), 7)) можно получить два различных результата в зависимости от того, определение какого шаблона мы применяем первым — F или M:

Сначала применяем F	F(John, M(h(v), 7)) = G(John, M(h(v), 7))
Сначала применяем M	F(John, M(h(v), 7)) = F(John, N(h(v), 7))

После применения второй подстановки получим тот же результат: G(John, N(h(v), 7)). Говорят, что этот набор подстановок *унифицирует* экземпляр шаблона как с F, так и с M.

Применение унификации в языке Prolog. Предположим, что у нас есть запрос q такой, что q = q_1, q_2, где q_1 и q_2 — подзапросы. Сначала мы пытаемся унифицировать q_1 с помощью какого-либо правила p из базы данных. Если запрос q_1 унифицируется с помощью p:

$$p :- p_1, p_2, \ldots, p_n$$

тогда мы можем создать новый запрос, подставляя $p\sigma$ вместо $q_1\sigma$:

$$p\sigma, q_2\sigma = p_1\sigma, p_2\sigma, \ldots, p_n\sigma, q_2\sigma = p_1', p_2', \ldots, p_n', q_2',$$

где апостроф (') обозначает исходный запрос, модифицированный преобразованиями σ. Теперь мы хотим разрешить этот новый запрос.

Однако если запрос унифицируется фактом r, тогда мы можем заменить q_1 на истину, и наш запрос теперь становится *истинным*, $q_2\sigma = q_2\sigma = q_2'$.

Тогда это является процессом выполнения для языка Prolog. Запросы унифицируются правилами или фактами из базы данных, пока в результате не получится *истина*. Конечно, если запрос q_1 унифицировался неподходящим правилом p или фактом r, то может получиться и *ложь*. В таком случае, если существует какая-либо альтернатива, ее надо проверять, пока не будет найдено верное решение. Набор преобразований α, необходимых для разрешения запроса, становится ответом на наш запрос. Основной принцип, применяемый здесь, называется принципом *резолюции* и более подробно будет описан в разделе 8.4.5.

Приведем более полный пример на языке Prolog — набор операторов (называемых *предложениями*) для определения сложения:

1. `succ(Y,X) :-Y is X+1.`
2. `add(Y,0,X) :-Y=X.`
3. `add(Z,X,Y) :-W is X-1, add(U,W,Y), succ(Z,U).`

Правило succ (successor — последующий элемент) вычисляет Y = X + 1 путем унификации переменной Y суммой X + 1. Таким образом, значение, подставляемое вместо Y, которое унифицирует Y и X + 1, есть значение выражения X + 1. (Следует помнить, что is вычисляет значение выражения, в то время как = обозначает невычисленную строку.) Правила add вычисляют сумму путем использования двух фактов о сложении:

0 + x = x (правило 2)

(x + 1) + y = x + (y + 1) = x + succ(y) (правило 3)

Правило 3 сводит сложение к выполнению одинакового количества шагов — «вычитание 1 из первого члена» и «прибавление 1 ко второму члену» — до тех пор, пока первый член не станет равным 0 (правило 2).

Например, для вычисления результата сложения 2 и 3, Y должен быть унифицирован в запросе add(Y,2,3). Prolog выдает следующий результат:

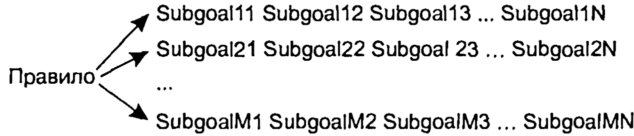
```
add(Y,2,3)
Y = 5
```

где Y = 5 — преобразование σ, которое унифицирует запрос с помощью правил базы данных, определяющих сложение.

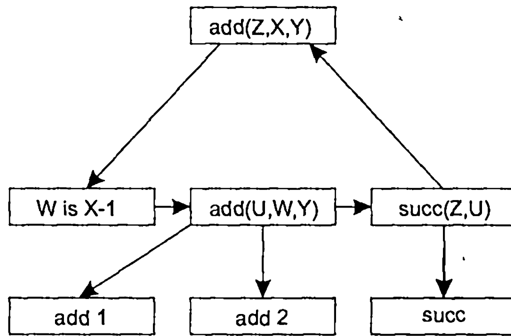
Реализация. Выполнение программы на языке Prolog иллюстрирует рис. 8.12. Оно представляет собой стандартный алгоритм обхода дерева. Набор фактов в базе данных языка Prolog (рис. 8.12, *а*) представляет собой набор из *M* целей, по одной для каждого правила, связанного с определенным отношением в базе данных. В примере со сложением add будет иметь две возможных цели: add1 с подцелью Y = X и add2 с подцелями W is X - 1, add(U,W,Y) и succ(Z,U) (рис. 8.12, *б*).

Для каждой цели (например, правила add2) Prolog последовательно пытается найти соответствие каждой подцели. Сначала W унифицируется с помощью X - 1. Эта операция присваивает W значение выражения X - 1. Затем Prolog пытается найти соответствие подцели add(U,W,Y), рекурсивно вызывая правила add (снача-

ла addl, затем add2 в поисках совпадения). Если совпадения найдены, то U становится суммой W и Y (то есть (X - 1) + Y). Если найдено соответствие с этой подцелью, тогда succ(Z,U) унифицирует Z с помощью U + 1 или Z устанавливается равным ((X - 1) + Y + 1) = (X + Y). Найдено соответствие с последней подцелью, и правило add2 выполнено путем унификации Z суммой X + Y.



а



б

Рис. 8.12. Выполнение программы на языке Prolog: *а* — база данных; *б* — унификация правила add2

Если соответствие с любой из подцелей не найдено, то применяется стандартный алгоритм отката. Предыдущая подцель проверяется на альтернативное соответствие. Если ни одно из соответствий не найдено, то проверяется еще одна предыдущая подцель и т. д. Если для всех подцелей соответствия не найдено, тогда правило не выполняется, и если есть какое-либо другое правило add, то выполняется оно.

Использование этого общего алгоритма обхода дерева позволяет вычислить add(Y. 2. 3) следующим образом (рис. 8.13):

1. add(Y, 2, 3) пытается найти соответствие правилу 3. W_1 принимает значение 2 - 1 = 1.
2. Prolog пытается унифицировать add(U_1, 1, 3), succ(Z_1, U_1).
3. add(U_1, 1, 3) устанавливает значение W_2 равным 0 и пытается унифицировать add(U_2, 0, 3), succ(Z_2, U_2).
4. add(U_2, 0, 3) выполняется путем унификации U_2 с 3 при использовании первого правила add.
5. succ(Z_2, U_2) = succ(Z_2, 3) выполняется путем унификации Z_2 с 4.

6. Z_2 (то есть 4) затем унифицируется с U_1 из отношения add(U1.1.3).
7. succ(Z1.U1) унифицирует Z_1 с U_1 + 1 = 4 + 1 = 5.
8. Наконец, Y унифицируется с Z_1 = 5, и это значение выводится на печать.

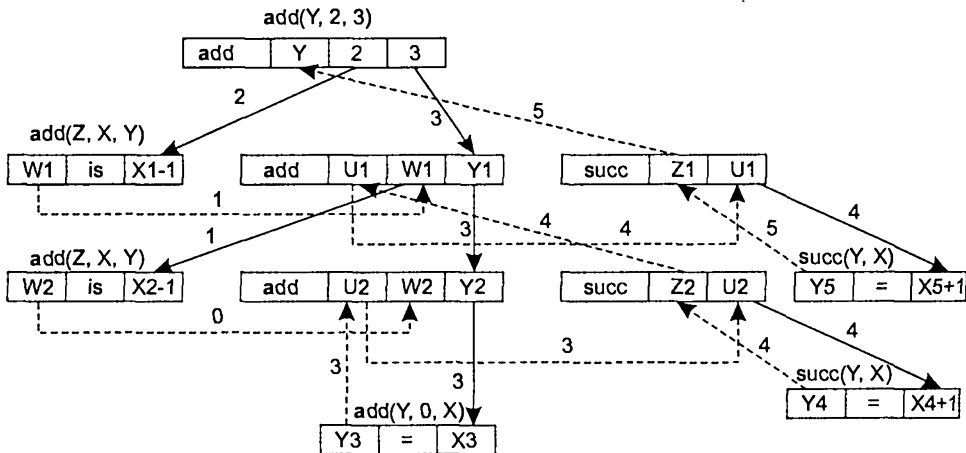


Рис. 8.13. Унификация в языке Prolog

Из этого примера должно быть видно, что стеки играют важную роль в реализации языка Prolog. Каждое предложение языка Prolog и каждая подцель сохраняются в стеке, пока не будет найдено соответствие цели.

8.4.4. Откат

При описании выполнения операции add на языке Prolog в предыдущем примере в процессе вычисления add(Y. 2. 3) было найдено соответствие со всеми подцелями. А что случится, если этого не произойдет? Что, если некоторая цель не сможет быть унифицирована с помощью правила из базы данных? В этом случае говорят, что правило не является логическим следствием программы.

Как показано на рис. 8.12, цели перечислены в некотором порядке 1...M. Но этот порядок произвольный, Prolog просто использует порядок, в котором факты записывались в базу данных. Пусть в некоторый момент времени мы пытаемся найти соответствие для некоторой подцели. Если соответствие найдено, то одна из возможных целей будет истинной, правда, мы не знаем, какая из них. В этом случае мы просто перебираем все возможные цели.

Если последняя из возможных целей также не привела к успеху, тогда говорят, что текущая подцель не выполнилась. Поскольку мы сохранили в стеке набор подцелей, среди которых производился поиск, мы возвращаемся к предыдущей подцели, соответствие с которой обнаружено, и пробуем найти соответствие с другой

возможной для нее целью. Мы называем описанный процесс общим алгоритмом *отката*, который можно описать, используя рис. 8.12, *б*.

1. Для подцели $Subgoal_{i,j}$ установить $k = 1$ в качестве новой цели.
2. Последовательно сопоставлять цель $goal_k$ для $k = 1..M$ и или добиться успеха, или нет в зависимости от того, выполнится ли какая-либо цель или все они не выполняются.
3. Если цель $goal_k$ выполняется, тогда выполняется и подцель $Subgoal_{i,j}$. Сохраняем k и ищем соответствие для $Subgoal_{i,j+1}$.
4. Если цель $goal_k$ ложна для всех k, значит, ложна и подцель $Subgoal_{i,j}$. Возвращаемся к подцели $Subgoal_{i,j-1}$ и пробуем для нее следующую цель $k + 1$.
5. Если найдено соответствие с подцелью $Subgoal_{i,N}$, то возвращаем в качестве результата для родительской цели $goal_k$ *истину*.
6. Если не найдено соответствие с подцелью $Subgoal_{i,j}$ ни для одного значения j, то возвращаем в качестве результата родительской цели $goal_k$ *ложь*.

Приведенный алгоритм в поисках правильного результата проверяет все возможные совпадения путем последовательного сохранения и удаления из стека частичных результатов. Этот часто используемый, хотя и медленный алгоритм лежит в основе различных поисковых стратегий.

Язык Prolog использует функцию ! (cut — вырезать) в качестве указателя на то, что не следует выполнять процедуру отката. Например, правило

```
A :-B,!,C,!,D.
```

выполнится только в случае, если будут найдены соответствия с первыми целями для B, C и D. Любая попытка найти соответствие с подцелью $Subgoal_{i,j-1}$ по предыдущему алгоритму, где $Subgoal_{i,j-1}$ =!, не увенчается успехом. Таким образом, если для D не будет найдено соответствия с первой целью, A также будет ложно, поскольку для C в качестве альтернативы не будет проверяться никакая новая цель. Использование функции ! в подходящих местах предупреждает ненужный во многих алгоритмах откат.

Откат — это общая методика программирования, доступная в любом языке, создающем древовидные структуры. Мы можем построить алгоритмы отката во всех языках, описанных в данной книге. Однако в таких языках, как LISP, где деревья являются естественным построением встроенного типа данных *список*, откат реализуется относительно легко. Как мы наблюдали ранее, в языке Prolog алгоритм отката является встроенным в сам язык.

8.4.5. Принцип резолюции

Исчисление предикатов сыграло важную роль в развитии языков, особенно языка Prolog. В 60-е гг. основным языком научных вычислений был FORTRAN, а для программ, основанных на логических выводах, преимущественно использовался LISP. Поскольку в языке LISP в качестве основного типа данных использовалась структура списков, деревья были естественным составным типом данных, которыми оперировали программы LISP. Если мы представляем правило $A\ B\ C \Rightarrow D$ в языке LISP, то естественной структурой данных будет дерево с корнем D и листьями A, B и C.

Для доказательства истинности D сначала требуется доказать истинность поддеревьев A, B и C. Как говорилось в разделе 8.4.4 при обсуждении алгоритма отката, LISP является естественным языком для построения таких алгоритмов.

В 1965 г. Робинсон (Robinson) разработал принцип резолюции, реализация которого в 1972 г. и стала языком Prolog. Хотя для программирования на языке Prolog нет необходимости полностью понимать принцип резолюции, тем не менее его знание помогает понять теорию, лежащую в основе выполнения программ на этом языке.

Пусть имеется множество предикатов $\{P_1, P_2, \ldots\}$. Основной задачей является определить, какую теорему можно генерировать из этих предикатов. Вообще, если P и Q предикаты, то является ли теоремой $P \Rightarrow Q$ (если P, то Q)?

Начнем с замечания, что любой предикат можно записать в *форме предложений*:

$$P_1 \land P_2 \land \ldots P_n \Rightarrow Q_1 \lor \ldots Q_m,$$

где P_i и Q_i — термы. Приведем примеры предложений.

Предикат	Предложение
$A \Rightarrow B$	$A \Rightarrow B$
$A \lor B$	$A \lor B \Rightarrow$ true
$A \lor B$	true $\Rightarrow A \lor B$
$\neg A$	$A \Rightarrow$ false
$A \lor \neg B \Rightarrow C$	$A \Rightarrow B \lor C$

Это можно показать путем приведения каждого предиката к *дизъюнктивной нормальной форме* (то есть последовательности термов $\lor$). Все термы с отрицанием становятся предпосылками, а термы без отрицания становятся следствиями предложений. Ниже приведена таблица с поясняющими примерами.

Дизъюнктивная нормальная форма	Предложение
$A \lor B \lor \neg C \lor D$	$C \Rightarrow A \lor B \lor D$
$\neg A \lor \neg B \lor \neg C \lor D$	$A \lor B \lor C \Rightarrow D$

Все это является прямым следствием следующих преобразований:

1. $P \Rightarrow Q$ эквивалентно $\neg P \lor Q$;
2. $\neg P \lor \neg Q$ эквивалентно $\neg(P \lor Q)$ (*закон Де Моргана*).

Сгруппировав вместе все термы с отрицанием в дизъюнктивной нормальной форме для предиката, в соответствии с законом Де Моргана можно утверждать, что они эквивалентны отрицанию всех предикатов, объединенных операцией И. В соответствии с преобразованием 1 они становятся предпосылкой в предложении.

Рассмотрим любое предложение:

$$P_1 \lor P_2 \lor \ldots P_m \Rightarrow Q_1 \lor Q_2 \lor \ldots Q_n.$$

Применение принципа резолюции для этого общего предиката становится слишком сложным. Но можно рассмотреть случаи, когда $n = 1$ или $n = 0$. Такие предикаты носят название *хорновых дизъюнктов*. Они имеют вид

$$P_1 \land P_2 \land \ldots P_m \Rightarrow Q_1;$$
$$P_1 \land P_2 \land \ldots P_m.$$

Если поменять местами предпосылки и следствия и заменить $\Rightarrow$ на :-, получим эквивалентные утверждения:

1) Q_1:- $P_1 \wedge P_2 \wedge \ldots P_m$;

2) :- $P_1 \wedge P_2 \wedge \ldots P_m$,

а это и есть известные нам предложения языка Prolog. Первое представляет правило, а второе — запрос. Если рассмотреть случай, когда $m = 0$, то получим факт языка Prolog Q_1:– true или просто

3) Q_1.

Тогда принцип резолюции для хорновых дизъюнктов теперь утверждает:

Пусть дан запрос $Q_1 \ldots Q_n$ и правило P_0:–$P_1 \ldots P_m$, тогда, если мы унифицируем Q_1 и P_0, или, как записывалось в разделе 8.4.3, $Q_1\sigma = P_0\sigma$, мы получим эквивалентные запросы.

Если мы унифицируем Q_1 с помощью P_0, то наш исходный запрос эквивалентен одному из следующих:

1) $(P_1 \ldots P_m Q_2 \ldots Q_n)\sigma = R_1 \ldots R_{m+n-1}$, если P_0 было правилом,

2) $(Q_2 \ldots Q_n)\sigma = R_1 \ldots R_{n-1}$, если P_0 было фактом (m=0),

где $R_i = P_j\sigma$ или $R_i = Q_j\sigma$. В конечном счете мы надеемся, что все R_i будут унифицированы фактами из базы данных и приведут к *истине* в процессе преобразований, дающих результаты по нашему запросу.

В этом и заключается суть выполнения программы на языке Prolog. Целью пространства поиска языка Prolog является унифицирование $Q_1 \ldots Q_n$, и Prolog свободен в выборе любого правила P из базы данных в качестве гипотезы, с помощью которой он постарается реализовать принцип резолюции. Если это приводит к успеху, то σ описывает ответы на наш запрос; в противном случае следует постараться использовать альтернативные правила P' для поиска правильной подстановки.

8.5. Рекомендуемая литература

Во многих книгах, упомянутых в главе 1, освещается вопрос управления последовательностью действий. В сентябрьском выпуске журнала *ACM Computing Surveys* [3] за 1989 г. напечатано несколько статей, посвященных парадигмам языков программирования, в которых рассмотрены необходимые структуры управления. Формальная связь между структурами управления и формальным доказательством корректности обсуждается в книге Ганнона [43]. Еще одна форма структур управления — таблица решений — описана у Мецнера и Барнеса [81], а в работе Praннера [89] представлена дополнительная информация по алгоритму отката.

8.6. Задачи и упражнения

1. Поясните, что будет происходить при выполнении фрагмента программы, написанной на языке С:

```
{int i=1
     j=2;
     if(i = j) {printf("true: %d %d\n",i,j);}
        else printf("false: %d %d\n",i,j);}
```

2. Четвертая форма синтаксиса выражений называется *обратной польской префиксной записью*, которая является обычной польской записью, записанной наоборот. Например, выражение a + (b × c) в префиксной записи будет выглядеть как + a × bc, в постфиксной — как abc × +, а в обратной постфиксной польской записи — как cb × a +. Свойства обратной польской префиксной записи аналогичны свойствам обычной постфиксной записи, но иногда она имеет некоторые преимущества перед обычной постфиксной записью при генерации эффективного машинного кода. Объясните почему.
3. Принимая во внимание порядок старшинства операций, принятый в языке C, предложите представления в виде дерева для следующих выражений:

   а) - A - B / C * D / E / F + G;

   б) A * B > C + D / E - F;

   в) ! A & B > C + D.

   Такое же задание выполните, принимая во внимание порядок старшинства операций в языке Pascal, а затем в Smalltalk.
4. Приведите блок-схемы всех первичных программ, состоящих из пяти узлов.
5. Покажите, что для любого $n > 1$ существует первичная программа, состоящая из n узлов.
6. Одной из основных теоретических предпосылок для замены операторов goto на структурные операторы является результат Бема (Bohm) и Якобини (Jacobini) [19]. Они доказали, что любая правильная программа может быть написана с помощью последовательностей одних только операторов if и while. Начертите схему правильной программы с несколькими циклами и ветвями. Запрограммируйте ее, используя операторы goto, а затем только конструкции if и while. Оцените оба эти варианта с точки зрения простоты их прочтения, понимания и записи.
7. Теорема Бема–Якобини утверждает, что любую блок-схему можно заменить на блок-схему, выполняющую те же функции, но в которой используются только операторы последовательного выполнения, а также if и while. Покажите, что оператор if не обязателен, то есть достаточно только оператора while.
8. Синтаксис оператора if языка Pascal следующий:

   ```
   if булево_выражение then оператор else оператор
   ```

   а на языке Ada оператор if выглядит следующим образом:

   ```
   if булево_выражение then оператор else оператор end if
   ```

   Поясните относительные достоинства наличия явного указателя на конец оператора, такого как end if в языке Ada.
9. Один из операторов цикла на языке Pascal имеет вид:

   ```
   for простая_переменная := начальное_значение to конечное_значение do оператор
   ```

   Обсудите достоинства и детали реализации, если:

   а) начальное_значение и конечное_значение вычисляются один раз, когда оператор for выполняется первый раз;

   б) начальное_значение и конечное_значение вычисляются каждый раз, когда программа начинает очередную итерацию цикла.

10. Правила языка Prolog можно рассматривать как логические предикаты. Правило a :- b, c приведет к тому, что a выполнится, если выполнятся и b, и c. Таким образом, можно сказать, что a истинно, если истинно условие $b \wedge c$.

    а) При каких условиях для p, q, r и s выполняются следующие правила Prolog?

1	2	3
x :- p.q.	x :- p.!.q.	x :- p.q.
x :- r.s.	x :- r.s.	x :- r.!.s.

    б) Выполните задание (а) с добавлением правила x:- fail после приведенных двух правил. Что будет, если это правило поместить первым в базе данных?

11. Правило append можно определить следующим образом:

```
append(X. Y. Z) :-X = []. Y = Z.
append(X. Y. Z) :-X = [A | B]. Z = [A | W]. append(B. Y. W).
```

    а) покажите, что это определение равносильно следующему:

```
append([]. X. X).
append( [H|X]. Y. [H|Z]) :- append(X. Y. Z).
```

    б) сделайте трассировку выполнения запроса append([1. 2. 3]. [4. 5]. Z), чтобы показать, что Z=[1.2.3.4.5] унифицирует запрос append.

Глава 9. Управление подпрограммами

В нескольких предыдущих главах мы описывали процесс, с помощью которого в языках программирования создаются типы данных и объекты данных этих типов. Мы рассмотрели, каким образом в программе выполняются последовательности операторов, позволяющие манипулировать такими данными, а также обсудили простые механизмы записей активаций, с помощью которых в языках осуществляется управление ресурсами памяти для различных объявляемых объектов данных. В этой главе мы более подробно рассмотрим взаимодействие подпрограмм и, что важнее, структурированные и эффективные способы обмена данными между подпрограммами.

Для большинства программ обычно достаточно простой стековой структуры для управления памятью при выполнении программы. Каждая новая процедура требует добавления к стеку нового блока памяти, содержащего локальные переменные для этой процедуры; выход из этой процедуры сопровождается изъятием этого блока из стека. В данной главе мы займемся изучением этой модели. Тем не менее необходим более развитый динамический механизм управления памятью. В языках С (с помощью функции `malloc`), С++, Java и Pascal (при помощи функции `new`) и неявно в таких языках, как ML и LISP программа может произвольным образом отводить в памяти место для новых объектов данных. Во всех таких случаях нужно использовать кучу, которая динамически выделяет в произвольном порядке блоки памяти и освобождает их по требованию. Организацию кучи мы обсудим более подробно в главе 10.

9.1. Управление последовательностью подпрограмм

Каким образом одна подпрограмма вызывает другую и затем позволяет вернуться из вызванной подпрограммы в исходную? Простая структура, состоящая из операторов `call` и `return`, является общей почти для всех языков программирования и представляет обычный стандартный механизм для достижения указанной цели.

Простой вызов и возврат из подпрограммы. В программировании мы привыкли рассматривать программы как некие иерархические структуры. Программа состоит из единственной главной программы, которая во время выполнения может вызывать различные подпрограммы, а те, в свою очередь, могут вызывать подпод-

программы и так далее до любого уровня глубины вложенности. Предполагается, что каждая подпрограмма в какой-то момент прекращает свое выполнение и возвращает управление той программе, которая ее вызвала. Во время выполнения подпрограммы выполнение вызывающей подпрограммы временно приостанавливается. Когда выполнение подпрограммы завершается, выполнение вызывающей программы возобновляется с той точки, которая непосредственно следует за точкой вызова подпрограммы. Такая структура управления часто объясняется *правилом копирования*: выполнение оператора `call`, вызывающего подпрограмму, имеет тот же эффект, что и замена этого оператора на копию тела вызываемой подпрограммы (с соответствующими подстановками для параметров и разрешением возможных конфликтов, связанных с совпадением идентификаторов) перед выполнением программы. С этой точки зрения подпрограмму можно рассматривать как структуру управления, которая позволяет избежать копирования большого количества идентичных или почти идентичных операторов, встречающихся в нескольких местах программы.

Прежде чем рассматривать реализацию простой структуры вызова-возврата, используемой для представления подпрограмм на основе правила копирования, следует обсудить вкратце некоторые неявные предположения, присутствующие в таком представлении, отказ от которых позволяет прийти к более общим структурам управления подпрограммами.

1. *Подпрограммы не могут быть рекурсивными*. Программа называется *прямо рекурсивной*, если в ней содержится обращение к самой себе (то есть подпрограмма `B` содержит оператор `call B`); *косвенно рекурсивная* подпрограмма содержит обращение к подпрограмме, которая, в свою очередь, вызывает первую подпрограмму или инициирует цепь вызовов подпрограмм, которая в конце концов возвращает к исходной подпрограмме.

   В случае, когда вызываемая подпрограмма является простой нерекурсивной подпрограммой, мы можем применить правило копирования во время трансляции, заменив вызовы этой подпрограммы на копии ее тела, и, таким образом, полностью исключить необходимость использования этой отдельной подпрограммы (в принципе, но не на практике). Но если подпрограмма является прямо рекурсивной, то это невозможно даже в принципе, поскольку подстановка вместо оператора вызова тела подпрограммы будет, очевидно, бесконечной. Каждая подстановка, удаляющая оператор вызова, вводит новый вызов той же самой подпрограммы, для которого необходима новая подстановка и т. д. Косвенная рекурсия, возможно, позволит удалить некоторые подпрограммы, но в конце концов должна привести к тому, что другие подпрограммы станут прямо рекурсивными. Тем не менее многие алгоритмы являются рекурсивными и, естественно, приводят к структурам рекурсивных подпрограмм.

2. *Требуется явное обращение к подпрограмме с помощью оператора* `call`. Для того чтобы можно было применить правило копирования, каждая точка вызова подпрограммы должна быть явным образом указана в транслируемой программе. Для подпрограммы, используемой в качестве *обработчика исключений*, может и не существовать явный оператор вызова.

3. *Подпрограммы должны полностью выполняться при каждом вызове.* В правиле копирования неявно предполагается, что подпрограмма выполняется с самого начала и до своего логического конца каждый раз, когда она вызывается. Если она вызывается во второй раз, то ее выполнение начинается заново и опять выполняется до своего логического конца вплоть до возвращения управления в вызывающую программу. Подпрограмма, которая используется как *сопрограмма,* продолжает выполнение с точки ее последнего завершения всякий раз, когда она вызывается.
4. *Немедленная передача управления в точку вызова.* Явный оператор вызова подпрограммы `call` указывает, что управление должно быть передано непосредственно этой подпрограмме в точке вызова, и, таким образом, копирование тела подпрограммы в вызывающую программу имеет тот же самый эффект. Для *вызова подпрограмм по графику* выполнение подпрограммы может быть отложено до некоторого более позднего момента времени.
5. *Единая последовательность выполнения.* В любой точке во время выполнения иерархии подпрограмм только одна подпрограмма имеет управление. Выполнение продолжается в одной последовательности от вызывающей программы к вызываемой подпрограмме и назад к вызывающей программе. Если в какой-либо точке мы остановим выполнение, мы всегда сможем определить, которая подпрограмма в данный момент выполняется (то есть имеет управление), а также указать множество других подпрограмм, выполнение которых временно было приостановлено (вызывающая программа, вызывающая ее программа и т. д.), и оставшиеся, которые или никогда не вызывались, или полностью были выполнены. Подпрограммы, используемые как *задачи*, могут выполняться одновременно, так что несколько подпрограмм могут выполняться в одно и то же время.

Из тех языков, которые мы обсуждаем в данной книге, только FORTRAN непосредственно основан на представлении подпрограмм с помощью правила копирования. В других языках используются более гибкие структуры управления.

9.1.1. Простые подпрограммы вызов-возврат

Заметим, что в данной главе мы уделяем особое внимание структурам управления последовательностью действий (то есть механизмам передачи управления от одних программ и подпрограмм другим). С каждой из этих структур тесно связаны вопросы управления данными: передача параметров, глобальные и локальные переменные и т. д. Эти темы отдельно обсуждаются в следующем разделе, а здесь мы полностью сосредоточим наше внимание на механизмах передачи управления. Например, даже в случае простой подпрограммы (без передачи параметров) вызов может происходить в двух формах: как *вызов функции* (для подпрограмм, которые в результате своего выполнения непосредственно возвращают некоторые значения) и как *вызов процедуры* или *подпрограммы* (для подпрограмм, результат действия которых выражается в побочном воздействии на общие данные).

Реализация. Для понимания реализации простой структуры управления вызов-возврат следует построить более полную модель того, что мы понимаем под

выполнением программы. Мы рассматриваем выражения и последовательности операторов как блоки выполняемого кода. Выполнение выражения или последовательности операторов означает просто выполнение кода с использованием аппаратно- или программно-моделируемого интерпретатора, как сказано в главе 2. Для выполнения подпрограмм требуется нечто большее:

1. Существует отличие между *определением* подпрограммы и *активацией* подпрограммы. Определение — это то, что мы видим в тексте программы и что при трансляции превращается в шаблон подпрограммы. Активация создается при каждом вызове подпрограммы с помощью шаблона, созданного на основе определения подпрограммы при трансляции.
2. Активация реализуется в виде двух частей: *сегмента кода*, куда входит выполняемый код и константы, и *записи активации*, содержащей локальные данные, параметры и различные другие объекты данных.
3. Сегмент кода *не изменяется* в процессе выполнения программы. Он создается транслятором и статически хранится в памяти. Он используется во время выполнения программы, но никогда не изменяется. Каждая активация данной подпрограммы использует один и тот же сегмент кода.
4. Запись активации *создается* заново каждый раз, когда вызывается подпрограмма, а по завершении подпрограммы запись активации разрушается. Во время выполнения подпрограммы содержимое записи активации постоянно меняется по мере того, как локальным переменным и другим объектам данных присваиваются новые значения.

Чтобы избежать путаницы, следует говорить не о «выполнении конкретного оператора S в подпрограмме», а о «выполнении оператора S во время активации R данной подпрограммы». Таким образом, чтобы отслеживать процесс выполнения программы, требуются два параметра, которые мы рассматриваем как два указателя, определяемых системой.

Указатель на текущую команду. Операторы и выражения, задействованные в подпрограмме, представлены определенными командами в исполняемом коде, который создается транслятором и хранится в сегменте кода. Мы считаем, что в любой момент во время выполнения программы существует некоторая команда в каком-либо сегменте кода, которая либо выполняется, либо сейчас начнет выполняться аппаратным или программно-моделируемым интерпретатором. Эта команда называется текущей командой, а указатель на нее хранится в специальной переменной, называемой *указателем на текущую команду* (current-instruction pointer, CIP). Действие интерпретатора сводится к выбору той команды, на которую указывает CIP, обновлению значения CIP (так, чтобы он указывал на следующую команду) и затем выполнению выбранной команды (которая может снова изменить CIP, выполняя переход на какую-либо другую команду).

Указатель текущей среды. Поскольку все активации подпрограммы используют один и тот же сегмент кода, простого знания текущей выполняемой команды недостаточно, необходимо также иметь указатель на используемую запись активации. Например, если какая-либо команда в коде ссылается на переменную X, эта переменная обычно представлена в записи активации. Каждая запись активации для этой подпрограммы имеет свой объект данных, именуемый идентификатором X.

Запись активации представляет собой среду ссылок подпрограммы, поэтому указатель на запись активации обычно известен как *указатель среды*. Указатель на текущую запись активации (текущую среду ссылок) в процессе выполнения программы хранится в переменной, которую мы называем *указателем на текущую среду* (current-environment pointer, CEP). Для того чтобы разрешить ссылку на X (то есть определить, какой именно объект данных понимается под X), и используется запись активации, на которую указывает CEP.

Теперь, когда в нашем распоряжении имеются указатели CIP и CEP, мы можем легко понять, как программа выполняется. Сначала создается запись активации для главной программы (поскольку эта запись активации существует в единственном экземпляре, она часто создается во время трансляции вместе с сегментом кода). Переменной CEP присваивается указатель на эту запись активации. Переменной CIP присваивается указатель на первую команду в сегменте кода главной программы. Интерпретатор начинает последовательно выбирать и выполнять команды, на которые ссылается указатель CIP.

Когда интерпретатор встречает команду вызова подпрограммы, создается запись активации и в переменную CEP помещается указатель на нее. Переменной CIP присваивается указатель на первую команду в сегменте кода вызываемой подпрограммы. С этого момента интерпретатор начинает выполнять команды подпрограммы. Если эта подпрограмма вызывает какую-либо другую подпрограмму, переменным CEP и CIP присваиваются новые значения, соответствующие указателям на запись активации очередной вызванной подпрограммы и первую команду ее сегмента кода.

Для корректного возвращения из вызванной подпрограммы в вызывающую значения CEP и CIP последней должны быть где-то сохранены командой вызова подпрограммы до того, как они будут изменены. Когда выполнение подпрограммы доходит до команды *возврата*, завершающей ее активацию, старые значения CEP и CIP, сохраненные при вызове этой подпрограммы, должны быть получены и восстановлены. Восстановление старых значений указателей — это все, что следует сделать для возвращения управления к корректной активации вызывающей подпрограммы в правильном месте, чтобы выполнение этой подпрограммы могло быть продолжено.

Где команда вызова подпрограммы должна сохранить значения переменных CIP и CEP перед тем, как присвоить им новые значения? Подходящим местом для этого является запись активации вызываемой подпрограммы. В эту запись активации включается дополнительный системный объект данных, называемый *точкой возврата*. Точка возврата содержит пространство для значений двух указателей, составляющих пару (ip, ep) — указатель команды и указатель среды (instruction pointer, environment pointer). После того как команда вызова подпрограммы создает запись активации, она сохраняет старые значения (ip, ep) указателей CIP и CEP в точке возврата, а самим указателям CIP и CEP присваиваются новые значения (ip, ep), что обеспечивает передачу управления вызываемой подпрограмме. Команда return извлекает старые значения (ip, ep) из точки возврата и восстанавливает значения CIP и CEP, возвращая, таким образом, управление вызывающей подпрограмме.

Теперь, если бы нам требовалось проследить за процессом выполнения программы в целом, мы бы увидели, что интерпретатор поочередно выполняет команды, на которые указывает CIP на каждом шаге цикла выполнения, используя CEP

для разрешения ссылок на данные (более подробно этот вопрос обсуждается в следующем разделе). Команды **вызова** подпрограммы и **возврата** из нее записывают значения (ip, ep) в переменные CIP и CEP и извлекают их оттуда, осуществляя таким образом передачу управления от одной подпрограммы другой. Если представить себе, что в какой-то момент выполнение программы приостанавливается, то, зная CIP и CEP, будет достаточно легко определить, какая подпрограмма выполнялась в данный момент, какая подпрограмма вызвала данную (эта информация содержится в точке возврата из подпрограммы) и т. д., пока не будет достигнута основная программа. На рис. 9.1 показана такая организация для главной программы и двух подпрограмм, каждая из которых вызывается из двух разных мест.

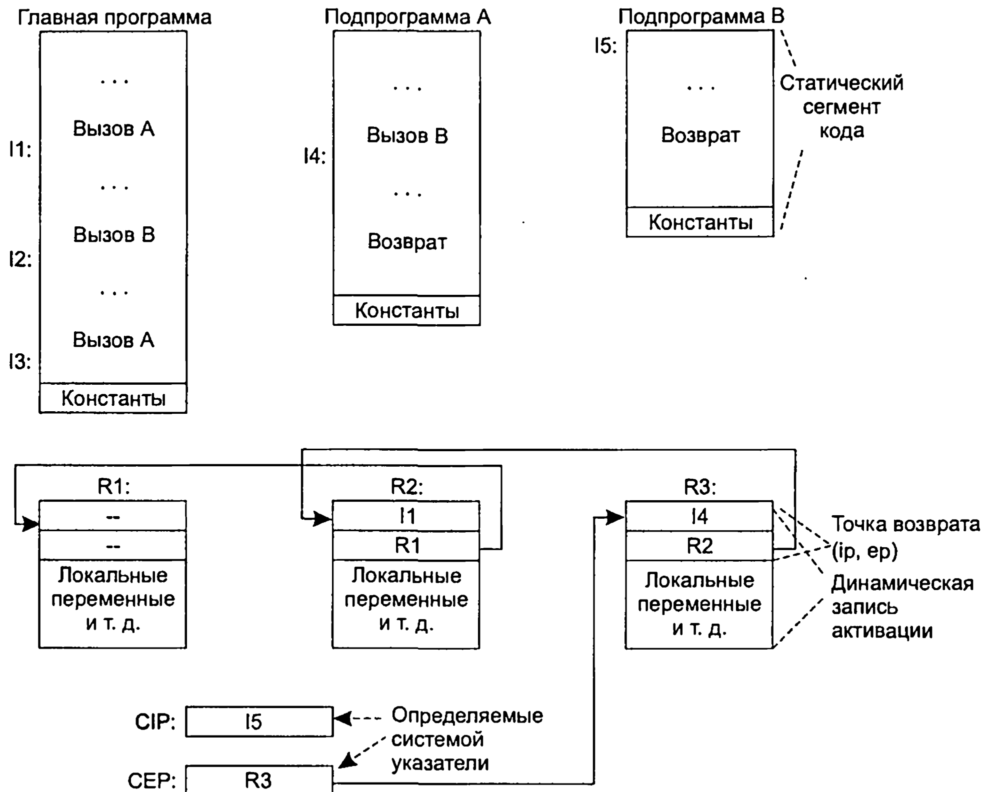


Рис. 9.1. Состояние выполнения в момент начала выполнения подпрограммы B

Эта модель реализации вызова подпрограммы и возврата из нее является достаточно общей, чтобы служить основой для нескольких разновидностей структур управления подпрограммами, которые мы рассмотрим позже. Отметим одну важную особенность подпрограмм, которые представляются с помощью правила копирования: в каждый момент выполнения программы может использоваться не более одной активации любой подпрограммы. Подпрограмма P может вызываться много раз в течение времени выполнения программы, но каждая активация подпрограммы завершается до начала следующей активации.

Основываясь на этом свойстве, можно вывести более простую модель реализации подпрограммы при условии, что за увеличение скорости мы готовы поплатиться объемом памяти. Эта более простая модель реализации заключается в том, чтобы статически отвести в памяти место для одной записи активации каждой конкретной подпрограммы как расширение сегмента кода, а не создавать запись активации во время вызова подпрограммы. В этой упрощенной модели (которая используется во многих реализациях языков COBOL и FORTRAN) уже в самом начале выполнения программы в памяти присутствуют сегменты кода и записи активации для всех подпрограмм и для главной программы. При выполнении программы не требуется динамически отводить место под запись активации при вызове подпрограммы. Вместо этого повторно используется одна и та же запись активации, которая просто инициализируется заново при каждом вызове подпрограммы. Поскольку в каждый момент времени используется только одна активация, такое многократное использование одной и той же ее записи для повторных вызовов подпрограммы не может разрушить необходимую информацию, оставшуюся от более раннего вызова, потому что все предыдущие вызовы уже завершены.

Располагая сегмент кода и запись активации в одном блоке памяти, мы можем произвести дальнейшие упрощения структуры управления. Теперь нам не нужен указатель CEP, поскольку запись активации является просто расширением сегмента кода, на который указывает CIP. Ссылка на переменную X в выполняемом коде легко разрешается при помощи присоединенной записи активации, которая, таким образом, позволяет обходиться без CEP. При отсутствии CEP нужно сохранять при вызове подпрограммы и восстанавливать при выходе из нее только указатель текущей команды CIP.

При использовании более общей реализации вызова подпрограммы и возврата из нее аппаратная часть компьютера может обеспечить лишь незначительную поддержку. Но для приведенной здесь упрощенной модели можно использовать часто предоставляемую аппаратурой команду перехода с возвратом для реализации вызова подпрограммы с помощью одной команды процессора. Указатель CIP в такой нашей модели представляется непосредственно с помощью аппаратного регистра адресов программы (см. главу 2). Команда перехода с возвратом сохраняет содержимое этого адресного регистра в некоторой области памяти или регистре (часто эта область памяти непосредственно предшествует той области, которой передается управление) и присваивает ссылку на указанную в ней область памяти в качестве нового значения адресного регистра (таким образом осуществляя передачу управления той команде, которая расположена в этой области памяти). В результате получается именно то, чего мы хотели: старое значение CIP сохранено, а его новым значением является адрес первой команды кода вызываемой подпрограммы. Возвращение из подпрограммы тоже, как правило, можно реализовать в виде одной аппаратной команды: сохраненное значение CIP снова присваивается адресному регистру (это делает аппаратная команда перехода). В результате мы получаем простую реализацию вызова подпрограммы и возврата из нее, правда, за это приходится расплачиваться увеличением используемого объема памяти, так как требуется заранее разместить в памяти записи активации для каждой подпрограммы. Пример такой структуры приведен на рис. 9.2.

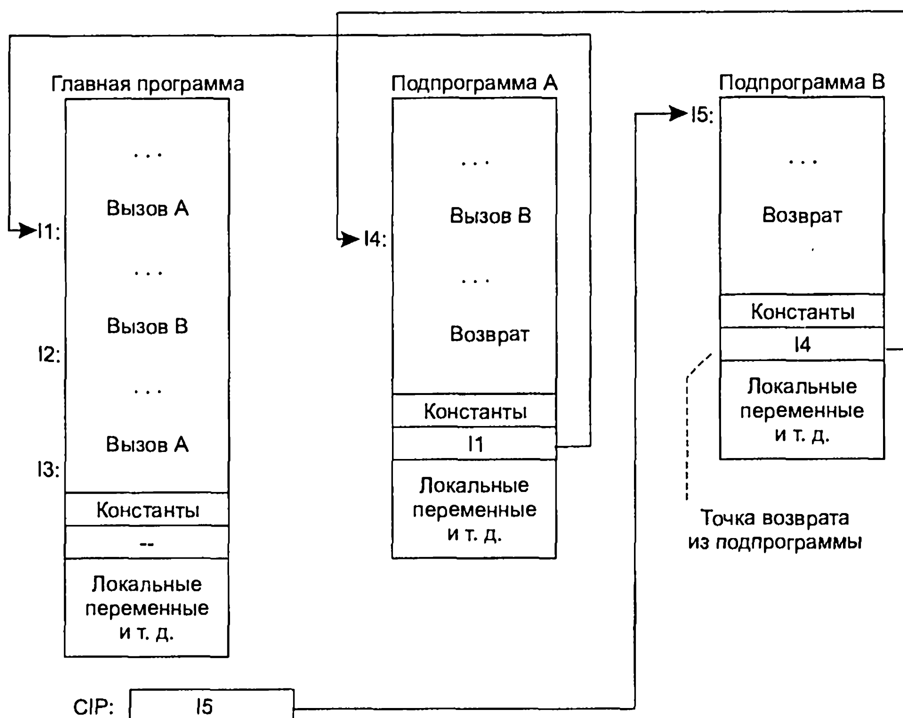


Рис. 9.2. Структура вызова-возврата из подпрограммы

Стековая реализация

Самым простым способом управления памятью для обработки структуры записей активации во время выполнения программы является стек. Свободная на начало выполнения программы память рассматривается как один последовательный блок в памяти. Когда в этом возникает необходимость, память выделяется из последовательности областей в этом стековом блоке, начиная с какого-то одного конца. Освобождается память в этом блоке в порядке, обратном его заполнению, так что освобождаемый блок памяти всегда расположен в вершине стека.

Для управления ресурсами памяти в таком стековом блоке требуется всего лишь один *стековый указатель*. Стековый указатель всегда указывает на вершину стека, следующее доступное слово свободной области в стековом блоке. Вся используемая в данный момент память в стеке расположена ниже того места в памяти, на которое указывает стековый указатель. Все свободное пространство расположено выше этого указателя. Когда требуется разместить блок из k минимально адресуемых компьютером областей памяти, указатель просто смещается на k областей дальше в свободном пространстве стека. Когда требуется освободить аналогичный объем памяти, то указатель смещается на k минимально адресуемых областей обратно в занятое пространство стека. *Уплотнение* происходит автоматически каждый раз, когда освобождается какой-либо объем памяти. Освобождение блока памяти автоматически приводит к его восстановлению как свободной области памяти и делает его доступным для повторного использования.

Большинство реализаций языка С построены на основе использования одного центрального стека записей активации подпрограмм и статически размещаемой области, содержащей системные программы и сегменты кодов подпрограмм. Структура типичной записи активации для С-подпрограммы показана на рис. 9.3, *а*. Запись активации содержит всю изменяемую информацию, связанную с активацией данной подпрограммы. На рис. 9.3, *б* показана типичная организация памяти во время выполнения программы на языке С (куча используется для размещения объектов, созданных при помощи функции new, которые затем уничтожаются при помощи функции dispose; обе эти функции более подробно обсуждаются в главе 10).

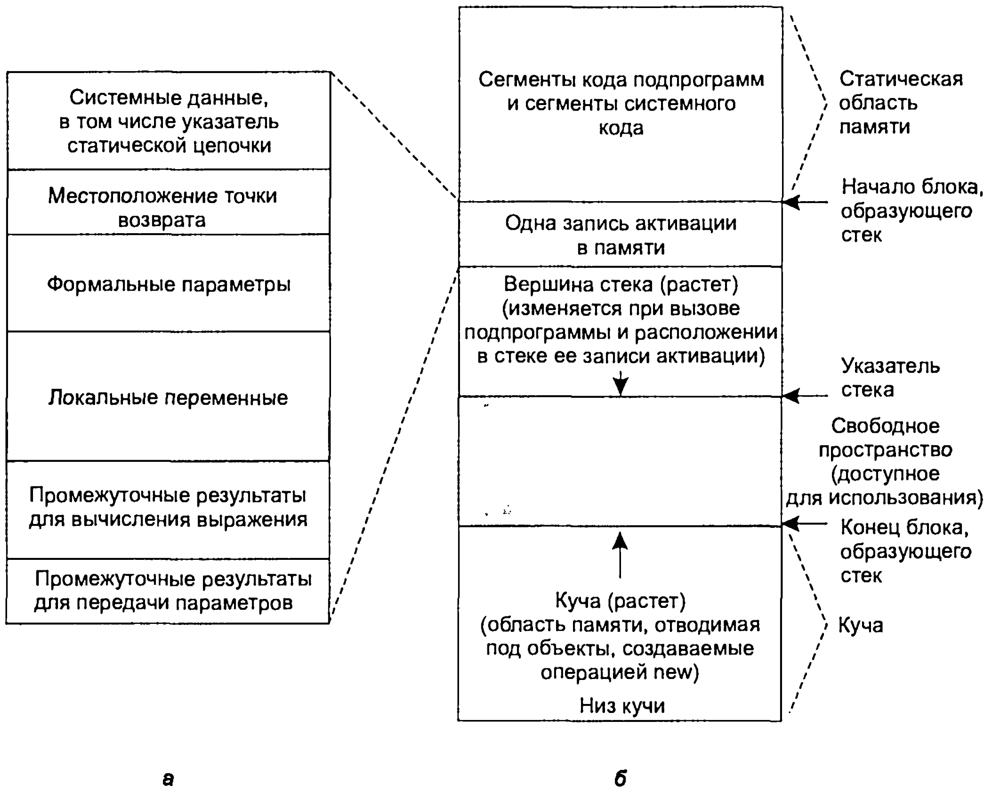


Рис. 9.3. Организация памяти в С: *а* — запись активации для подпрограмм; *б* — организация памяти во время выполнения

Использование стека в реализации языка LISP несколько иное. Вызовы подпрограмм (функций) также являются строго вложенными, а для хранения записей активации также может использоваться стек. Каждая запись активации содержит точку возврата и области памяти для временного хранения значений, необходимых для вычисления выражений и передачи параметров. Локальные среды ссылок могут быть также размещены в том же самом стеке, но отличие заключается в том, что программисту позволено непосредственно манипулировать этими ассоциациями. Вследствие этого они, как правило, хранятся в отдельном стеке, пред-

ставленном как связанный список, называемый *А-списком*. Тогда стек, содержащий точки возврата и временные значения, может быть скрыт от программиста и заполняться последовательно. Для реализации языка LISP также необходима куча, которая управляется посредством списка свободной памяти и процессом сборки мусора. Типичная организация памяти в языке LISP представлена на рис. 9.4.

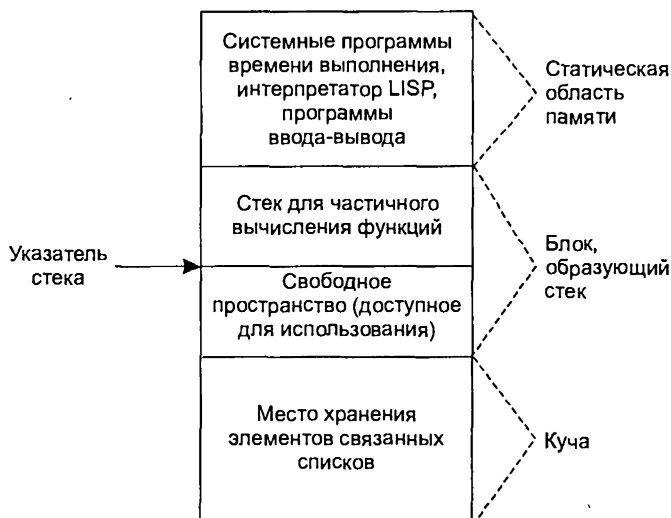


Рис. 9.4. Организация памяти во время выполнения для LISP

9.1.2. Рекурсивные подпрограммы

Вернемся к одному из основных предположений, сделанных нами относительно используемых подпрограмм, — отсутствию рекурсии — и попытаемся разработать идеологию языка, который позволял бы использовать рекурсивные подпрограммы. Рекурсия в форме вызовов рекурсивных подпрограмм — это одна из наиболее важных структур управления в программировании. Большое количество алгоритмов наиболее естественным образом представляется с использованием рекурсии. В языке LISP, где первичной доступной структурой данных являются списковые структуры, рекурсия является первичным механизмом управления для повторяющихся последовательностей операторов, заменяя циклы, характерные для большинства других языков.

Спецификация. Если мы допускаем вызов рекурсивной подпрограммы, то подпрограмма A может вызывать любую подпрограмму, в том числе и A, и подпрограмму B, содержащую вызов A и т. д. При написании программы это не требует никаких изменений в синтаксисе, так как вызов рекурсивной подпрограммы ничем не отличается от вызова обычной подпрограммы. Концептуально здесь также нет никаких сложностей при условии, что мы четко представляем себе отличие определения подпрограммы от ее активации. Единственное различие между рекурсивным вызовом и обычным вызовом заключается в том, что рекурсивный вызов создает вторую активацию этой же подпрограммы *во время жизни ее первой акти-*

вации. Если вторая активация подпрограммы приводит к следующему рекурсивному вызову, то три активации могут существовать одновременно и т. д. Вообще говоря, если при выполнении программы образуется цепь, состоящая из первого вызова подпрограммы A, за которым следуют k рекурсивных вызовов этой же подпрограммы, происходящие до того, как произошло хотя бы одно возвращение из A, то перед возвращением из k-й рекурсивно вызванной подпрограммы будет существовать $k + 1$ активаций подпрограммы A. Единственным нововведением, связанным с рекурсией, является, таким образом, одновременное сосуществование нескольких активаций одной и той же подпрограммы.

Реализация. Поскольку мы хотим обеспечить возможность одновременного сосуществования нескольких активаций, нам нужны оба указателя — CEP и CIP. В момент очередного вызова подпрограммы создается новая запись активации, которая впоследствии разрушается при выходе из нее.

Обратите внимание: на рис. 9.1 ничто не указывает на то, что новые записи активации следует создавать для *уникальных* подпрограмм A и B. Находясь во время выполнения в подпрограмме A, мы могли бы с такой же легкостью создать новую запись активации для самой же подпрограммы A, как и для подпрограммы B. В случае, когда A вызывает B, их *времена жизни не могут перекрываться*; для любых двух активаций подпрограмм A и B время жизни A *полностью включает в себя* время жизни B. Отсюда следует, что если подпрограмма A вызовет не B, а рекурсивно саму себя, то отмеченное свойство активаций будет справедливо и в этом случае, и новая запись активации для A может быть также добавлена в стек, содержащий более раннюю запись активации подпрограммы A.

Каждая запись активации содержит точку возврата, в которой хранятся значения пары (ip, ep), используемые операторами call и return. Если на рис. 9.1 рассмотреть только значения ep, сохраняемые в точках возврата, можно заметить, что они образуют связный список, связывающий вместе записи активации в центральном стеке в порядке их создания. С помощью указателя CEP можно получить доступ к записи активации, содержащейся на вершине центрального стека. По значению указателя ep, содержащегося в точке возврата этой записи активации, можно получить доступ ко второй записи активации в центральном стеке; указатель ep этой второй записи активации содержит ссылку на третью и т. д. В конце этой цепи связей находится указатель на запись активации главной программы. Эта цепь связей называется *динамической цепью*, так как она связывает вместе активации подпрограмм в порядке их динамического создания во время выполнения программы. (В разделе 9.4.2 обсуждается близкое понятие — *статическая цепь*, которая связывает вместе записи активации в целях получения ссылок.)

Аппаратная часть традиционного компьютера иногда предоставляет некоторую поддержку для рассмотренной организации с центральным стеком, но, как правило, ее используют для реализации чего-нибудь более дорогостоящего, чем реализация простой структуры вызова-возврата без рекурсии. Не представляет сложности использовать подпрограммы, реализованные этим простым способом, совместно с подпрограммами, использующими центральный стек, при условии, что компилятор умеет различать эти подпрограммы при компиляции команд call и return. Только те подпрограммы, которые действительно вызываются рекурсивно, нуждаются в реализации с использованием центрального стека. Так, в некото-

рых языках, например в PL/I, подпрограммы, вызываемые рекурсивно, должны помечаться как RECURSIVE в своих объявлениях; в других языках, подобных C и Pascal, всегда подразумевается рекурсивная структура.

9.1.3. Объявление forward в языке Pascal

Рекурсия в вызове процедур создает проблему при использовании стратегии одного прохода, применяемой при разработке многих компиляторов Pascal. Пусть, например, A и B являются подпрограммами и A вызывает B, а B вызывает A. Тогда, если определение подпрограммы A появляется перед определением B, то естественно, что вызов подпрограммы B в A появляется до определения подпрограммы B. Поменяв местами определения подпрограмм A и B, мы не решим эту проблему, а лишь обратим ее. Эта проблема решается в Pascal, если использовать для подпрограммы, определяемой в последнюю очередь, *объявление* forward, имеющее вид сигнатуры подпрограммы, включая полный список параметров, за которым вместо тела подпрограммы следует слово forward. В качестве примера можно привести следующее объявление:

```
procedure A(formal_parameter_list); forward;
```

С помощью такого опережающего объявления для подпрограммы A подпрограмма B может быть определена полностью, а затем дается полное определение тела подпрограммы A (но список формальных параметров не повторяется). Опережающее объявление подпрограммы A с использованием ключевого слова forward дает компилятору достаточно информации, чтобы корректно компилировать вызов A, содержащийся в подпрограмме B, хотя полное описание подпрограммы A еще не появилось. Это тот же принцип, который используется в спецификации пакетов Ada (package), где сигнатура подпрограммы дается без деталей ее реализации.

Поскольку список параметров не должен повторяться, когда подпрограмма позже определяется, его отсутствие является основным источником ошибок программирования. В этом случае рядом с листингом тела подпрограммы мы нигде не обнаружим документацию сигнатуры подпрограммы. Неправильно используемые параметры, а также некорректно используемые передаваемые по имени и ссылке параметры в теле подпрограммы могут вызвать серьезные проблемы. Одним из способов избежать их является включение в определение подпрограммы комментария, содержащего список параметров. Поэтому после опережающего объявления подпрограммы с помощью спецификации forward фактическое определение тела подпрограммы может быть дано в следующем виде:

```
procedure A {(список формальных параметров)};
      begin
      ...
      end
```

Причина того, что в языке Pascal так акцентируется необходимость определения идентификатора до его использования, кроется в ошибочном мнении, что для эффективной компиляции необходим однопроходный компилятор. За один проход по исходному тексту программы такой компилятор получает весь спектр необходимой информации, считывая и обрабатывая определение одной подпрограммы за один раз и генерируя выполняемую объектную программу, как только он

завершил чтение подпрограммы. Для обработки программы за один проход компилятор в каждой точке программы должен иметь достаточно информации о значении каждого идентификатора, чтобы сгенерировать корректный объектный код. Как говорилось в главе 2, для уменьшения времени компиляции не обязательно использовать однопроходные компиляторы, а сегодня при наличии дешевых и быстрых микропроцессоров время компиляции больше не является столь значимой проблемой.

Листинг 9.1. Аномалия с опережающим объявлением подпрограммы в Pascal с использованием спецификации forward

```
program anomaly(input,output);
procedure S; {1}
      begin
      writeln('wrong one')
      end;
procedure T;
      {Здесь пропущено: procedure S; forward;}
      procedure U;
         begin
         S {2}
         end;
      procedure S; {3}
         begin
         writeln('right one')
         end;
      begin
      U
      end;
begin
T
end.
```

Правило, касающееся употребления конструкции forward в Pascal, приводит к странной аномалии, которая проиллюстрирована в программе anomaly (листинг 9.1). Возможны три различных интерпретации этой программы.

1. Компиляция не будет выполнена, потому что программа anomaly неправильна. Вызов процедуры S в точке {2} вызывает процедуру S, определенную в точке {3}, что является опережающей ссылкой без декларации forward.
2. Вызов процедуры S в точке {2} вызывает процедуру S, определенную в точке {1}, — как раз то, что сделал бы однопроходный компилятор, хотя это неправильная процедура S в контексте области видимости этого вызова процедуры.
3. Программа выполняется, вызывая в точке {2} процедуру S, определенную в точке {3}. Это правильная процедура, которую можно вызывать, даже если требуемая, хотя и избыточная, спецификация forward отсутствует.

Внимательное чтение стандарта языка Pascal [70] поможет определить правильную интерпретацию.

1. Раздел 6.2.2.1 стандарта утверждает, что *определяющей точкой* для S{3} является второй оператор procedure S.
2. Разделы 6.2.2.2 и 6.2.2.3 определяют область для определяющей точки S{3} как всю процедуру T. Следовательно, вызов S{2} является вызовом S{3}, а не S{1}.

3. Раздел 6.2.2.9 требует, чтобы определяющая точка (например, оператор S{3}) для любого фактического идентификатора (например, S{2}) появлялась раньше его использования, что неявно подразумевает необходимость использования объявления forward.

Имея такое подробное объяснение, мы легко приходим к выводу, что первая интерпретация, очевидно, правильна, в то время как третья, хотя и неверна, является обоснованной интерпретацией. Вторая интерпретация, очевидно, неверна и является наихудшим вариантом из трех.

После выполнения этой программы с помощью тринадцати различных трансляторов языка Pascal от нескольких фирм-поставщиков для различных компьютеров и операционных систем, включая PC, Macintosh, рабочие станции UNIX, были получены следующие результаты:

Интерпретация 1 (правильная)	3 компилятора
Интерпретация 2 (худший вариант)	7 компиляторов
Интерпретация 3 (неправильная, но обоснованная)	3 компилятора

Очевидно, что лишь незначительное меньшинство следовало стандарту, а больше половины продавцов выбрали необоснованную интерпретацию (по понятным причинам мы не раскрываем здесь их имена). (См. задачу 22 в конце этой главы.)

Этот странный пример можно еще несколько усложнить. Если вернуться к вопросам стандартизации языков, обсуждавшимся в разделе 1.3.3, то можно прийти к выводу, что все 13 компиляторов действовали фактически в соответствии со стандартом. Дело в том, что компилятор должен обрабатывать в соответствии со стандартом те программы, которые сами по себе являются соответствующими стандарту (а данная программа к ним не относится), поэтому в данном случае для компилятора не существует определенных указаний, что ему делать. Следовательно, с точки зрения соответствия стандарту все три интерпретации верны.

9.2. Атрибуты управления данными

Управление данными в языках программирования связано с организацией доступа к данным во время выполнения программы в различных ее точках. Механизмы управления последовательностью действий, описанные в предыдущей главе, предоставляют средства координирования последовательности выполнения операций во время выполнения программы. Когда доходит очередь до какой-либо операции, ей требуется предоставить данные, над которыми она выполняет определенные действия. Возможности управления данными в языке программирования определяют, каким образом могут быть предоставлены необходимые для выполнения операции данные и как результат выполнения одной операции может быть сохранен, а позже использован в качестве операнда следующей операции.

При написании программы программист обычно хорошо представляет себе, какие операции и в какой последовательности должны быть выполнены в программе, но гораздо реже он может сказать то же самое об операндах этих операций. Например, в программе на языке С может содержаться следующий оператор:

```
X := Y + 2 * Z.
```

Простой анализ показывает, что в нем последовательно выполняются три операции: умножение, сложение и присваивание. Но что можно сказать об операндах этих операций? Ясно, что одним из операндов операции умножения является число 2, но остальные операнды только обозначены идентификаторами X, Y и Z, которые, очевидно, сами по себе не являются операндами, а лишь некоторым образом указывают на операнды. Идентификатор Y может обозначать вещественное или целое число или имя подпрограммы, не имеющей параметров, выполнение которой приводит к вычислению операнда. Возможно, программист допустил ошибку и Y на самом деле обозначает булеву величину, или строку символов, или вообще служит меткой какого-либо оператора. Y может обозначать как некоторую величину, вычисленную недавно, например в предыдущем операторе, так и величину, вычисленную гораздо раньше и отделенную от операции присваивания, где используется Y, многими уровнями вызовов подпрограмм. Ситуация станет еще сложнее, если идентификатор Y используется в разных частях программы для обозначения различных объектов. Какое из определений Y следует использовать в данном месте программы?

В двух словах, центральная проблема управления данными — это проблема определения того, что означает Y при каждом выполнении такого оператора присваивания. Поскольку Y может быть как локальной, так и не локальной переменной, эта проблема включает в себя то, что известно как области видимости для объявленных имен. Поскольку Y может быть формальным параметром, имеют значение способы передачи параметров; а так как Y может быть именем подпрограммы, не имеющей параметров, — то и механизмы возврата результатов из подпрограмм. В следующем подразделе мы сначала обсудим, что собой представляют перечисленные задачи, а в разделе 9.3 более подробно опишем методы их реализации.

9.2.1. Имена и среды ссылок

Существует два способа, с помощью которых объект данных можно сделать доступным в качестве операнда некоторой операции.

- *Непосредственная передача.* Объект данных, вычисленный в определенной точке программы как результат выполнения некоторой операции, может быть непосредственно передан другой операции в качестве ее операнда (например, результат умножения 2 × Z непосредственно передается в качестве операнда операции сложения в операторе X := Y + 2 × Z). В этом случае объект данных временно располагается в памяти на все время его жизни и может даже никогда не получить имени.

- *Ссылка через именованный объект данных.* При создании объекта данных ему может быть присвоено имя, которое затем используется для указания того, что этот объект является операндом некоторой операции. С другой стороны, объект данных может быть компонентом некоторого другого объекта, имеющего имя, так что имя этого более крупного объекта может быть использовано совместно с операцией выбора для указания исходного объекта данных как операнда какой-либо операции.

Непосредственная передача используется для управления данными в выражениях, но для управления данными вне выражений в большинстве случаев исполь-

зуются имена объектов данных и ссылки на них. Определение того, какой объект скрывается под некоторым именем, является центральной проблемой управления данными.

Элементы программы, которые могут иметь имена

Какие типы имен можно встретить в программах? В каждом языке они свои, но существуют некоторые общие для многих языков категории типов имен, перечисленные ниже:

1) имена переменных;
2) имена формальных параметров;
3) имена подпрограмм;
4) имена типов данных, определяемых программистом;
5) имена констант;
6) метки операторов (имена для операторов);
7) имена исключений;
8) имена элементарных операций (например, +, *, SQRT);
9) имена буквальных констант (например, 17, 3.25).

Здесь мы в основном будем заниматься первыми тремя категориями имен — именами переменных, формальных параметров и подпрограмм. Что касается остальных категорий, то большинство ссылок на имена этих групп разрешаются во время трансляции программы, а не во время ее выполнения, как уже говорилось ранее. Если разобраться в основных идеях, лежащих в процессе управления данными для первых трех категорий, то распространить их на остальные категории не представляет большого труда.

Имя, относящееся к любой из перечисленных категорий, называется *простым именем*. *Составное имя* — это имя компонента некоторой структуры данных. Оно записывается как простое имя, обозначающее всю структуру целиком, за которым следует последовательность одной или более операций выбора, которые выбирают конкретный элемент данной именованной структуры. Например, если A — это имя массива, то A является простым именем, а A[3] — составное имя. Составные имена могут быть достаточно сложными (например, A[3].Class[2].Room). В большинстве языков простые имена представлены идентификаторами вида X, Z2, Sub1, поэтому термины «имя» и «идентификатор» являются взаимозаменяемыми.

Ассоциации и среды ссылок

К управлению данными по большей части относятся процедуры связывания идентификаторов (простых имен) с конкретными объектами данных и подпрограммами. Такое связывание называется *ассоциацией* и может быть представлено как пара, состоящая из идентификатора и связанного с ним объекта данных (или подпрограммы).

Во время выполнения программы мы можем наблюдать следующее.

1. В начале выполнения главной программы ассоциации идентификаторов связывают имя каждой объявленной в ней переменной с конкретным объек-

том данных, а каждое имя подпрограммы, вызываемой в главной программе, — с конкретным определением подпрограммы.

2. Когда выполняется главная программа, она вызывает *операции обработки ссылок* для определения конкретного объекта данных или подпрограммы, ассоциированной с идентификатором. Например, для выполнения оператора присваивания

   ```
   A := B +FN(C)
   ```

   необходимо выполнить четыре операции обработки ссылок для получения объектов данных, ассоциированных с именами A, B и C, и подпрограммы, ассоциированной с именем FN.
3. При вызове каждой новой подпрограммы для нее создается новое множество ассоциаций. Имя каждой переменной и каждого формального параметра, объявленных в подпрограмме, ассоциируется с определенным объектом данных. Также могут быть созданы новые ассоциации для имен подпрограмм.
4. Когда выполняется подпрограмма, она вызывает операции обработки ссылок для определения конкретного объекта данных или подпрограммы, ассоциированных с каждым идентификатором. Некоторые из этих ссылок могут быть ссылками на ассоциации, созданные при входе в подпрограмму, в то время как другие могут указывать на ассоциации, созданные ранее в главной программе.
5. Когда подпрограмма возвращает управление в главную программу, ее ассоциации уничтожаются (или становятся неактивными).
6. Когда управление возвращается в главную программу, выполнение продолжается, как и ранее, при этом используются первоначально созданные в начале выполнения главной программы ассоциации.

В этом шаблоне создания, использования и уничтожения ассоциаций мы видим основные концепции управления данными.

Среды ссылок. В каждой программе и подпрограмме имеется множество ассоциаций идентификаторов, доступных для разрешения ссылок во время их выполнения. Это множество ассоциаций идентификаторов называется *средой ссылок* подпрограммы (или программы). Среда ссылок подпрограммы обычно не меняется во время ее выполнения. Она устанавливается в момент создания активации подпрограммы и остается неизменной в течение всего времени жизни активации. Значения различных объектов данных могут меняться, но ассоциации имен с объектами данных и подпрограмм остаются неизменными. Среда ссылок подпрограммы может состоять из нескольких компонентов.

1. *Среда локальных ссылок* (или просто *локальная среда*). Множество ассоциаций, созданных при входе в подпрограмму и представляющих формальные параметры, локальные переменные и подпрограммы, определенные только внутри этой подпрограммы, формирует *среду локальных ссылок* этой активации подпрограммы. Значение ссылки на имя из локальной среды можно определить, не выходя за пределы активации подпрограммы.
2. *Среда нелокальных ссылок.* Множество ассоциаций для идентификаторов, которые могут использоваться в подпрограмме, но не были созданы при входе

в нее, называется *средой нелокальных ссылок* (или просто *нелокальной средой*) подпрограммы.

3. *Среда глобальных ссылок.* Если ассоциации, созданные в начале выполнения главной программы, доступны для использования в подпрограмме, то они формируют *среду глобальных ссылок* (или просто *глобальную среду*) этой подпрограммы. Среда глобальных ссылок является частью среды нелокальных ссылок.
4. *Среда предопределенных ссылок.* Для некоторых идентификаторов существует предопределенная ассоциация, которая задается непосредственно в определении языка. Любая программа или подпрограмма может использовать подобные ассоциации без явного их создания.

Видимость. Говорят, что ассоциация для идентификатора *видима* в подпрограмме, если она является частью среды ссылок для этой подпрограммы. Существующая, но не входящая в среду ссылок выполняющейся в данный момент подпрограммы ассоциация, называется *скрытой* от этой подпрограммы. Часто ассоциация для идентификатора становится скрытой при входе в подпрограмму, которая переопределяет идентификатор, уже используемый где-то в программе.

Динамическая область видимости. Каждая ассоциация имеет *динамическую область видимости*, которая ассоциируется с тем периодом выполнения программы, в течение которого она существует как часть среды ссылок. Таким образом, динамическая область видимости ассоциации состоит из множества активаций подпрограмм, внутри которых она видима.

Операции обработки ссылок. Операция обработки ссылки — это операция с сигнатурой:

```
операция_обработки_ссылки : идентификатор × среда_ссылок → объект_данных или подпрограмма
```

где операция_обработки_ссылки при заданных идентификаторе и среде ссылок находит соответствующую ассоциацию для этого идентификатора в этой среде ссылок и возвращает связанный с идентификатором объект данных или определение подпрограммы.

Локальные, нелокальные и глобальные ссылки. Ссылка на идентификатор является *локальной ссылкой*, если операция обработки ссылки находит ассоциацию в локальной среде; ссылка является *нелокальной* или *глобальной ссылкой*, если ассоциация находится в *нелокальной* или *глобальной* среде соответственно. (Термины *нелокальный* и *глобальный* часто используются как взаимозаменяемые для обозначения любой ссылки, не являющейся локальной.)

Псевдонимы для объектов данных

В течение своего времени жизни объект данных может иметь более одного имени (то есть может существовать несколько ассоциаций в разных средах ссылок, каждая из которых по-разному именует объект данных). Например, когда объект данных предается по ссылке (раздел 9.3) в некоторую подпрограмму как ее параметр, в этой подпрограмме на него можно ссылаться с помощью имени соответствующего формального параметра, и в то же время он сохраняет свое первоначальное имя в вызывающей программе. С другой стороны, объект данных может стать компонентом нескольких объектов данных через организацию связи при помощи указателей и, таким образом, иметь несколько составных имен, которые можно исполь-

зовать для доступа к нему. Почти все языки программирования предлагают разнообразные способы задания различных имен для одного и того же объекта данных.

Когда объект данных видим *в одной среде ссылок* и имеет не одно имя (простое или составное), то каждое из этих имен называется *псевдонимом*[1] для этого объекта данных. Если объект данных имеет несколько имен, но в каждой среде ссылок, в которой он появляется, у него только одно имя, то никаких проблем не возникает. Но возможность ссылки на один и тот же объект в одной и той же среде ссылок с использованием разных имен приводит к серьезным проблемам как для пользователя, так и для разработчика языка. В листинге 9.2 приведены две программы на языке Pascal, в которых целочисленная переменная в различных местах программы имеет два имени — J и I. В первой программе использование псевдонимов не встречается, так как ни в какой момент выполнения оба имени J и I не могут использоваться в одной и той же подпрограмме. Во второй программе в подпрограмме Sub1 имена I и J являются псевдонимами для одного и того же объекта данных, так как I передается по ссылке в подпрограмму Sub1, где оно становится связанным с именем J, но в то же время I также видимо в подпрограмме Sub1 как нелокальное имя.

Листинг 9.2. Использование псевдонимов в программе на языке Pascal: *а* — без пседвонимов, *б* — I и J являются псевдонимами в Sub1

```
а) program main(output);
     procedure Sub1(var J: integer);
        begin
        ...
                 { J видимо, I невидимо }
        end;
     procedure Sub2;
        var I: integer;
        begin
        ...
        Sub1(I);  { I видимо, J невидимо }
        ...
        end;
     begin
     ...
     Sub2         {Оба имени невидимы}
     ...
     end.
б) program main(output);
     var I: integer;
     procedure Sub1(var J: integer);
        begin
        ...
                 { I и J ссылаются на один и тот же объект данных }
        end;
     procedure Sub2;
        begin
        ...
        Sub1(I);  { I видимо, J невидимо }
        ...
        end;
     begin
```

[1] В литературе псевдоним иногда называется термином «алиасное имя» (от англ. alias). — *Примеч. науч. ред.*

```
...
Sub2        { I видимо. J невидимо }
...
end.
```

Использование псевдонимов создает трудности для программиста, поскольку затрудняет понимание программы. Например, если в программе вы встретите следующую последовательность операторов:

```
X := A + B:
Y := C + D:
```

то присваивания переменным X и Y, очевидно, независимы друг от друга и могут быть расположены в любом порядке; если переменная X далее в программе вообще не используется, то первое присваивание можно полностью удалить. Однако предположим, что X и C являются псевдонимами для одного и того же объекта данных. Тогда эти операторы присваивания являются взаимозависимыми, и попытка поменять их местами или удалить первый оператор приведет к трудно обнаруживаемой ошибке. Возможность использования псевдонимов усложняет верификацию программы, так как ни о каких двух переменных нельзя с уверенностью сказать, что они заведомо ссылаются на разные объекты данных.

Кроме того, использование псевдонимов вызывает аналогичные проблемы и для разработчика. Частью процесса оптимизации программного кода во время трансляции является реорганизация или удаление ненужных шагов вычислений. Если допускается использование псевдонимов, эта часть оптимизации становится невозможной без дополнительной проверки того, что два фрагмента вычисления, внешне совершенно независимых друг от друга, не связаны между собой через псевдонимы. Из-за описанных проблем, вызываемых использованием псевдонимов, в некоторых новых языках иногда предпринимаются попытки ограничить или вовсе запретить те свойства, которые позволяют создавать псевдонимы.

Пример 9.1. Ссылки на переменные в языке Pascal

В листинге 9.3 приведена простая программа на языке Pascal с помеченной средой ссылок для каждой подпрограммы. Обратите внимание на то, что каждый из идентификаторов A, C и D объявлен в двух местах. Идентификатор A является именем формального параметра в SUB1, а также объявлен как имя переменной в главной программе. Идентификатор C является именем формального параметра в SUB2 и также именем переменной в главной программе. D является именем локальной переменной в обеих подпрограммах SUB1 и SUB2. Тем не менее в каждой среде ссылок видимой является только одна ассоциация для каждого из этих имен. Так, в подпрограмме SUB2 видимой является локальная ассоциация для C, а глобальная ассоциация для C в главной программе скрыта. В операторе C := C + B подпрограммы SUB2 используется локальная ссылка на переменную C и глобальная ссылка на переменную B, объявленную в главной программе.

Среда предопределенных ссылок здесь не показана. В языке Pascal она состоит из постоянных типа MAXINT (максимально допустимое целое значение) и подпрограмм типа read, write и sqrt. Для каждого из этих предопределенных имен можно создать новую ассоциацию при помощи явного объявления и, таким образом, сделать невидимой предопределенную ассоциацию для части программы.

Листинг 9.3. Среды ссылок в программе на языке Pascal

```
program main:
var A. B. C: real:
procedure Sub1(A: real):
      var D: real:
      procedure Sub2 (C:real):
         var D: real:
         begin                              Среда ссылок для Sub2
         - Операторы                        Локальные     C. D
         C := C+B:                          Нелокальные   A. Sub2 в Sub1
         - Операторы                                      B. Sub1 в main
         end:
      begin                                 Среда ссылок для Sub1
      - Операторы                           Локальные     A. D. Sub2
      Sub2(B):                              Нелокальные   B. C. Sub1 в main
      - Операторы
      end:
begin
- Операторы                                 Среда ссылок для main
Sub1(A):                                    Локальные:    A. B. C. Sub1
- Операторы
end.
```

9.2.2. Статическая и динамическая области видимости

Динамическая *область видимости* ассоциации для идентификатора, как было сказано в предыдущем разделе, представляет собой то множество активаций подпрограмм, в которых эта ассоциация является видимой во время выполнения программы. Динамическая область видимости ассоциации всегда включает в себя активацию подпрограммы, в которой эта ассоциация была создана как часть среды локальных ссылок. Она также может быть видимой как нелокальная ассоциация в активациях других подпрограмм.

Правило динамической области видимости определяет динамическую область видимости каждой ассоциации в терминах динамических изменений, возникающих при выполнении программы. Например, типичное правило динамической области видимости утверждает, что область видимости ассоциации, созданной во время активации подпрограммы P, включает не только саму эту активацию, но и любую активацию подпрограммы, вызванной подпрограммой P или подпрограммой, вызванной подпрограммой P и т. д., до тех пор, пока активация вызванной позже подпрограммы не определит новую локальную ассоциацию для идентификатора, которая скроет исходную активацию. С помощью этого правила динамическая область видимости ассоциации связывается с *динамической цепью* активаций подпрограмм, описанной в разделе 9.1.2.

Когда мы смотрим на *текст программы*, то замечаем, что организация ассоциации ссылок на идентификаторы с конкретными объявлениями или определениями смысла этих идентификаторов также представляет некоторую проблему. Например, в листинге 9.3 ссылки на имена B и C в операторе присваивания C := C + B в подпрограмме SUB2 должны быть связаны с конкретными объявлениями имен C и B как переменных или формальных параметров. Но какими объявлениями? Каж-

дое объявление или другое определение идентификатора в пределах текста программы имеет определенную область видимости, называемую его *статической областью видимости*.

Для простоты будем считать, что термин *объявление* используется здесь для ссылки на объявление переменных, определение подпрограммы, определение пользовательского типа данных, определение константы или другие средства определения смысла для конкретного идентификатора, встречающегося в тексте программы. Объявление создает в тексте программы ассоциацию между идентификатором и некоторой информацией об объекте данных или о подпрограмме, именем которого или которой и будет служить данный идентификатор во время выполнения программы. *Статическая область видимости* объявления — это та часть текста программы, где использование идентификатора является ссылкой на это конкретное объявление идентификатора. *Правило статической области видимости* — это правило для определения статической области видимости объявления. В языке Pascal, например, правило статической области видимости используется для определения того, что ссылка на переменную X в подпрограмме P отсылает к объявлению переменной X в начале подпрограммы P, а если такого объявления там нет, то к объявлению X, расположенному в начале подпрограммы Q, содержащей подпрограмму P и т. д.

Правила статической области видимости сопоставляют ссылки с объявлениями в тексте программы; правила динамической области видимости сопоставляют ссылки с ассоциациями для имен во время выполнения программы. Как должны быть связаны эти правила? Очевидно, *что они должны быть согласованы* друг с другом. Например, если правила статической области видимости в языке Pascal сопоставляют ссылку на переменную B в операторе `C := C + B` (см. листинг 9.3) с объявлением имени B в главной программе, то правила динамической области видимости также должны сопоставить ссылку на B во время выполнения программы с объектом данных, названным B в главной программе. В тексте программы может быть несколько объявлений для B, а во время выполнения программы — несколько объектов данных, названных B в различных активациях подпрограмм во время выполнения. Таким образом, поддержание *согласованности* между правилами статической и динамической области видимости является не слишком тривиальной задачей. Существует несколько способов ее решения, которые мы рассмотрим ниже.

Важность статической области видимости. Предположим, что какой-то язык не использует правила статической области видимости. Рассмотрим оператор вида `X := X + Max` из какой-либо подпрограммы. Без использования правил статической области видимости во время трансляции ничего невозможно определить относительно имен X и Max. Во время выполнения программы, когда очередь доходит до выполнения данного оператора, операция обработки ссылок сначала должна найти соответствующие ассоциации для X и Max, а затем должны быть определены тип и другие атрибуты имен X и Max. Существует ли ассоциация для каждого идентификатора? Является ли Max именем подпрограммы, именем переменной, меткой оператора, именем типа или именем формального параметра? Если X — имя переменной, принадлежит ли эта переменная к такому типу, который можно складывать с Max? На эти вопросы невозможно ответить, пока не будет установлено, ссылкой на какие объекты являются имена X и Max. Более того, каждый раз при

выполнении этого оператора весь процесс требуется повторить заново, поскольку ассоциации X и Max могли измениться с момента предыдущего выполнения оператора. В языках LISP, SNOBOL4 и APL правила статической области видимости почти не используются. Таким образом, ссылка на любое имя в процессе выполнения программ на этих языках вызывает инициирование довольно сложного и дорогостоящего процесса интерпретации, который сначала ищет соответствующую ассоциацию для указанного имени (если таковая вообще существует), а затем определяет тип и атрибуты ассоциированного с именем объекта данных или подпрограммы.

Правила статической области видимости позволяют выполнить этот процесс для большинства ссылок на имена, встречающиеся в программе, только один раз во время трансляции, вместо того чтобы выполнять его многократно во время выполнения программы. Например, если в программе на языке Pascal встречается оператор присваивания X := X + Max и где-то в программе имя Max определяется как константа с помощью объявления const Max = 30, то правила статической области видимости позволяют во время трансляции связать ссылку на Max с этим (или каким-то другим) объявлением имени Max. Затем компилятор Pascal может определить, что при выполнении нашего оператора присваивания значение Max всегда равно 30, и оттранслировать оператор в исполняемый код, в котором к X просто добавляется 30 без выполнения операции обработки ссылки для имени Max. Аналогично, если правила статической области видимости языка Pascal позволяют связать ссылку на X с объявлением X : real где-то в тексте программы, тогда компилятор Pascal может выполнить статическую проверку типов, то есть он может определить, что при выполнении этого оператора:

1) будет существовать ассоциация, связывающая имя X и некоторый объект данных;
2) этот объект данных будет относиться к типу вещественных чисел (real);
3) его значение будет типом, который можно использовать в качестве аргумента операции сложения.

Из объявления X компилятор не сможет определить ни *местоположение* в памяти объекта данных, на который ссылается имя X (поскольку местоположение определяется динамически во время выполнения программы и может быть различным для различных случаев выполнения оператора), ни *значение* X (поскольку оно также определяется динамически во время выполнения программы). Тем не менее статическая проверка типов позволяет значительно ускорить выполнение программы и повысить надежность ее работы (поскольку во время трансляции ошибки определения типов отслеживаются для всех ветвей программы).

Правила статического определения области видимости позволяют установить много разных типов связей между ссылками на имена и их объявлениями во время трансляции. Два из них были упомянуты ранее: связывание имени переменной с объявлением переменной и связывание имени константы с ее объявлением. К другим типам связей относится связывание имен типов с объявлениями типов, связывание формальных параметров со спецификациями формальных параметров, связывание вызовов подпрограмм с объявлениями подпрограмм и связывание меток операторов, используемых в операторах goto, с метками конкретных опе-

раторов. В каждом из этих случаев во время трансляции могут быть сделаны многочисленные упрощения, которые повысят эффективность выполнения программы.

Правила статической области видимости также важны для программиста при чтении программы, поскольку они позволяют связать имя, используемое в программе, с объявлением для этого имени без необходимости отслеживать процесс выполнения программы. Например, правила статической области видимости языка Pascal позволяют связать ссылку на переменную X в каком-либо операторе с расположенным где-то в программе объявлением X без какого-либо анализа последовательности вызовов подпрограмм, начиная с главной программы и заканчивая выполнением данного оператора. Таким образом, правила статической области видимости упрощают понимание программы.

9.2.3. Блочная структура

Концепция *блочной структуры*, используемая в *блочно-структурированных языках* Pascal, PL/I и Ada, заслуживает специального упоминания. Программы, написанные на блочно-структурированных языках, имеют характерную структуру и соответствующий набор правил статической области видимости. Впервые эти понятия появились в языке ALGOL 60 — одном из наиболее важных ранних языков программирования. Благодаря своей элегантности и влиянию на эффективность реализации они были приняты и в других языках.

В блочно-структурированном языке каждая программа или подпрограмма организована как множество вложенных блоков. Главной характеристикой *блока* является то, что он вводит новую среду локальных ссылок. Блок начинается с множества объявлений имен (объявления переменных, определения типов, определения констант и т. д.), за которым следует множество операторов, в которых имеются ссылки на указанные имена. Для простоты мы будем считать блок эквивалентным объявлению подпрограммы, хотя точное определение блока различно в различных языках. Объявления, расположенные в блоке, определяют его среду локальных ссылок. Эта локальная среда не меняется во время выполнения операторов, составляющих тело блока. В языке C имеется блочная структура, но она существует только в пределах одной подпрограммы. Это дает возможность определять нелокальные имена без каких-либо накладных расходов, связанных с активацией подпрограммы. Позже мы еще вернемся к этому вопросу.

Вложение блоков осуществляется путем размещения определения одного блока полностью внутри некоторого другого блока. На самом верхнем (или внешнем) уровне программа состоит из одного блока, определяющего главную программу. Внутри этого блока располагаются другие блоки, определяющие подпрограммы, которые вызываются непосредственно из главной программы; внутри этих блоков содержатся другие, в которых определяются подпрограммы, вызываемые из подпрограмм первого уровня и т. д. В листинге 9.4 представлена типичная компоновка блочно-структурированной программы. В таких языках, как C и Ada, самый верхний уровень может состоять из нескольких независимых блоков (каждый из которых содержит внутри себя вложенные блоки и может компилироваться по отдельности), но здесь мы ограничимся рассмотрением только одного внешнего блока.

Листинг 9.4. Статическая блочная структура программы

```
program Main;
 - Локальные обьявления для Main;                 Начало Main
 procedure Subl;                                            |
  - Локальные обьявления для Subl;           Начало Subl    |
  procedure Sub3;                                      |    |
     - Локальные обьявления для Sub3; Начало Sub3      |    |
     begin                                         |   |    |
     - Операторы для Sub3;                         |   |    |
     end {Sub3};                             Конец Sub3    |    |
  procedure Sub4;                                      |    |
    - Локальные обьявления для Sub4;  Начало Sub4      |    |
    begin                                          |   |    |
    - Операторы для Sub4;                          |   |    |
    end {Sub4};                              Конец Sub4    |    |
  begin                                                |    |
  - Операторы для Subl                                 |    |
  end {Subl};                                  Конец Subl   |
 procedure Sub2;                                            |
  - Локальные обьявления для Sub2            Начало Sub2    |
  begin                                                |    |
  - Операторы для Sub2                                 |    |
  end {Sub2};                                  Конец Sub2   |
 begin                                                      |
 - Операторы для Main;                                      |
end {Main }.                                       Конец Main
```

Правила статической области видимости для блочно-структурированных программ выглядят следующим образом.

1. Объявления в заголовке блока определяют среду локальных ссылок для блока. Любая ссылка на идентификатор в пределах тела блока (не включая вложенные подблоки) рассматривается как ссылка на локальное объявление идентификатора (если оно присутствует).

2. Если в пределах тела блока имеется ссылка на идентификатор, но локальное объявление идентификатора отсутствует, то эта ссылка рассматривается как ссылка на объявление в первом блоке, который включает в себя данный блок. Если и в нем искомое объявление идентификатора отсутствует, то поиск продолжается в блоке следующего уровня и т. д. Если же объявление не будет найдено и в самом внешнем (верхнем) блоке, то идентификатор либо относится к предопределенным именам данного языка и его объявление берется из среды предопределенных имен, либо (если такого имени нет и среди предопределенных имен) эта ситуация рассматривается как ошибка. Таким образом, среда предопределенных имен языка рассматривается как блок, внешний по отношению к самому внешнему блоку программы.

3. Если блок содержит определение другого блока, то все локальные объявления в этом внутреннем блоке или блоках, содержащихся внутри него, полностью скрыты от внешнего блока, в котором, следовательно, не могут появиться ссылки на эти определения. Таким образом, внутренние блоки инкапсулируют свои локальные объявления, делая их невидимыми для внешних блоков.

4. Блок может иметь имя (обычно если он представляет собой именованную подпрограмму). Имя блока входит в среду локальных ссылок *содержащего блока*. Например, если в главной программе на Pascal содержится определение подпрограммы, которое начинается следующим образом:

```
procedure P(A: real);
```

то имя процедуры P является локальным именем в главной программе, тогда как имя формального параметра A является частью среды локальных ссылок самой подпрограммы P. В пределах главной программы можно ссылаться на P, но не на A.

Обратите внимание на то, что, используя эти правила статической области видимости, можно несколько раз объявлять один и тот же идентификатор в различных блоках, но объявление во внешнем блоке всегда становится скрытым внутри вложенного блока, если в последнем этот идентификатор объявляется заново.

Эти правила статической области видимости для блочно-структурированных программ позволяют связывать каждую ссылку в пределах одного блока с определенным объявлением этого имени в процессе трансляции подпрограммы (если ссылка не является ошибочной). Это не требует от программиста никаких явных действий, за исключением создания правильных объявлений в пределах каждого блока и правильного вложения блоков друг в друга. На основе правил статического контроля компилятор языка может провести статическую проверку типов и некоторые другие упрощения структуры выполняемого кода программы. Все это послужило причиной того, что блочная структура была принята в качестве структуры программы во многих языках программирования.

9.2.4. Локальные данные и среды локальных ссылок

Теперь мы начинаем рассмотрение различных структур управления данными, используемых в языках программирования. В этом разделе мы остановимся на средах локальных ссылок, которые образуют простейшую структуру. В следующих разделах рассмотрим нелокальные среды, параметры и передачу параметров.

Локальная среда подпрограммы Q состоит из различных идентификаторов, объявленных в заголовке подпрограммы Q (но имя подпрограммы не входит в эту среду). Имена переменных, формальных параметров и подпрограмм образуют среду локальных ссылок подпрограммы Q. Здесь имеются в виду те подпрограммы, которые локально определены внутри Q (то есть подпрограммы, определения которых вложены в Q).

Для сред локальных ссылок правила динамической и статической области видимости легко согласовываются между собой. Правило статической области видимости гласит, что ссылка на идентификатор X в теле подпрограммы Q связывается с локальным объявлением X в заголовке подпрограммы Q (в предположении, что такое объявление существует). Правило динамического контроля определяет, что ссылка на X во время выполнения подпрограммы Q для своего разрешения использует ассоциацию идентификатора X в текущей активации подпрограммы Q (заметим, что, вообще говоря, может существовать несколько активаций подпрограммы Q, но толь-

ко одна из них может выполняться в настоящий момент). Для реализации правила статической области видимости компилятор просто создает и поддерживает таблицу локальных объявлений идентификаторов, расположенных в заголовке подпрограммы Q, а в процессе компиляции ее тела в первую очередь обращается именно к этой таблице, когда ему требуется найти объявление какого-либо идентификатора.

Правило динамической области видимости может быть реализовано двумя способами, каждый из которых задает разную семантику локальных ссылок. Рассмотрим подпрограммы P, Q и R (листинг 9.5). В подпрограмме Q объявлена локальная переменная X. Подпрограмма P вызывает подпрограмму Q, которая, в свою очередь, вызывает подпрограмму R. По завершении подпрограммы R управление передается обратно в Q, которая, завершаясь, снова передает управление подпрограмме P. Проследим за переменной X во время выполнения этой последовательности обращений к подпрограммам.

1. Когда выполняется P, переменная X не видна из P, так как она является локальной переменной подпрограммы Q.
2. Когда подпрограмма P вызывает подпрограмму Q, переменная X становится видимой как имя целочисленного объекта данных с начальным значением 30. При выполнении подпрограммы Q ее первый оператор ссылается на X и печатает ее текущее значение 30.
3. Когда подпрограмма Q вызывает подпрограмму R, ассоциация для идентификатора X становится скрытой, но сохраняется в течение всего времени выполнения R.
4. Когда подпрограмма R возвращает управление Q, ассоциация для идентификатора X снова становится видимой. X по-прежнему является именем того же самого объекта данных, который по-прежнему имеет значение равное 30.
5. Когда подпрограмма Q возобновляет свое выполнение, переменная X увеличивается на 1 и ее новое значение 31 выводится на печать.
6. Когда подпрограмма Q возвращает управление подпрограмме P, то ассоциация для X снова становится скрытой, но для нее можно предусмотреть два различных действия:
   - *Сохранение.* Ассоциация X может быть сохранена до следующего вызова подпрограммы Q точно так же, как это происходило при вызове подпрограммы R. В этом случае при следующем вызове Q идентификатор X остается связан с тем же объектом данных, значение которого по-прежнему равно 31. Следовательно, первый оператор подпрограммы Q выведет на печать значение 31. Если этот цикл повторится и подпрограмма Q будет вызвана в третий раз, то X будет иметь значение 32 и т. д.
   - *Уничтожение.* Альтернативой сохранению ассоциации является ее удаление (то есть ассоциация, связывающая X с объектом данных, разрушается, так же как и сам объект данных, а место в памяти освобождается для использования). Когда Q вызывается во второй раз, создается новый объект данных, которому присваивается начальное значение 30, и его ассоциация с именем X создается заново. В этом случае первый оператор подпрограммы Q при ее вызове всегда выводит на печать одно и то же значение 30.

Листинг 9.5. Среда локальных ссылок: сохранение или уничтожение?

```
procedure R;
        ...
end;
procedure Q;
var X: integer := 30;                  -начальное значение X равно 30
begin
        write (X);                     -вывод на печать значения X
        R;                             -вызов подпрограммы R
        X := X + 1;                    -увеличение значения X на 1
        write (X)                      -повторный вывод X на печать
end;
procedure P;
        ...
        Q;                             -вызов подпрограммы Q
        ...
end;
```

Сохранение и уничтожение — это два различных подхода к семантике среды локальных ссылок, и оба они связаны с понятием *времени жизни* среды. В языках Java, C, Pascal, Ada, LISP APL и SNOBOL4 используется подход, основанный на уничтожении ассоциации: локальные переменные не сохраняют свои значения в промежутках между последовательными вызовами подпрограммы. В COBOL и многочисленных версиях FORTRAN используется подход, основанный на сохранении ассоциации: переменные сохраняют свои старые значения между вызовами подпрограммы. В PL/I и ALGOL предусмотрены оба подхода; для каждой конкретной переменной можно выбрать свой вариант.

Реализация

При обсуждении реализации сред ссылок удобно представлять локальную среду подпрограммы как *таблицу локальной среды*, состоящей из пар «идентификатор–объект данных» (см. табл. 9.1 для программы, представленной в листинге 9.6). Как уже было сказано, размер выделяемой каждому объекту памяти осуществляется в соответствии с его типом (см. главу 6), а местоположение объекта данных в памяти определяется с помощью его *l*-значения (см. главу 5). Приведенный способ представления таблицы локальной среды вовсе не означает, что фактические идентификаторы (например, X) хранятся во время выполнения программы именно таким образом. Обычно это не так. Имя используется только для того, чтобы последующие ссылки на эту переменную были способны определить, в каком месте памяти располагается эта переменная во время выполнения программы. С использованием таблиц локальных сред реализация локальных сред на основе подходов с сохранением и уничтожением ассоциаций становится элементарной.

Листинг 9.6. Определение подпрограммы Sub в языке Pascal

```
procedure SUB(X:integer);
    var Y: real;
       Z: array [1..3] of real;
    procedure SUB2;
       begin
       ...
       end {Sub2};
    begin
    ...
    end {Sub};
```

Таблица 9.1. Таблица локальной среды для подпрограммы Sub

Имя	Тип	Содержимое *l*-значения
X	integer	Определяется параметром
Y	real	Локальная величина
Z	real	Локальный массив
		Дескриптор: [1..3]
SUB2	procedure	Указатель на сегмент кода

Сохранение локальных сред. Если локальная среда подпрограммы Sub из листинга 9.6 должна быть сохранена в промежутках между вызовами, тогда одна таблица локальной среды, содержащая сохраняемые переменные, размещается в *сегменте кода* подпрограммы Sub как некоторая его часть (рис. 9.5).

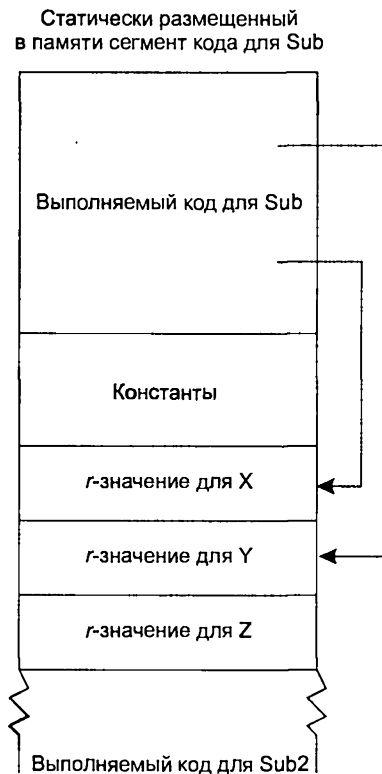


Рис. 9.5. Размещение сохраняемых локальных переменных и ссылки на них

Поскольку сегменту кода память выделяется статически и он всегда доступен в процессе выполнения программы, все переменные, расположенные в той части этого сегмента, который отведен под таблицу локальной среды, также сохраняются во время выполнения программы. Если переменной присвоено начальное значение, например значение 30 для Y, то это начальное значение может быть сохранено в объекте данных, когда ему отводится некоторое место в памяти (аналогично

тому, как было бы сохранено значение константы в сегменте кода). Если предположить, что каждая сохраняемая переменная объявлена в начале подпрограммы Sub, то в таком случае компилятор сможет определить размер каждой переменной из таблицы локальной среды и вычислить смещение от начала сегмента кода (базовый адрес) до начала области памяти, выделяемой под объект данных. Когда какой-либо оператор из сегмента кода ссылается на переменную Y во время выполнения, смещение этой переменной добавляется к базовому адресу сегмента кода для определения местоположения объекта данных, ассоциированного с именем Y. Сам идентификатор Y не нужен во время выполнения, поэтому он вообще не сохраняется.

Такая реализация сохранения для локальной среды не требует каких-либо специальных действий для сохранения значений объектов данных; значения объектов, сохраненные на момент завершения вызова подпрограммы Sub, будут находиться в тех же областях памяти и при следующем ее вызове. Также не требуется каких-либо специальных действий для перехода от одной локальной среды к другой, когда одна подпрограмма вызывает другую. Поскольку код и локальные данные для каждой подпрограммы являются частью одного и того же сегмента кода, передача управления сегменту кода другой подпрограммы автоматически приводит к переходу на среду локальных ссылок этой вызванной подпрограммы.

Удаление локальных сред. Если локальная среда подпрограммы Sub подлежит удалению между вызовами и должна создаваться заново при каждом входе в подпрограмму, тогда таблица локальной среды, содержащая удаляемые переменные, должна храниться в памяти как часть записи активации подпрограммы Sub. Предположим, что запись активации при входе в подпрограмму Sub создается в центральном стеке и удаляется из него при выходе из подпрограммы, как описано в разделе 9.1, тогда удаление среды локальных ссылок происходит автоматически. В предположении, что каждая подлежащая удалению локальная переменная объявляется в начале определения подпрограммы Sub, компилятор опять-таки сможет определить количество переменных и размер каждой переменной, содержащейся в таблице локальной среды, и вычислить смещение начала области размещения каждого объекта данных относительно начала записи активации (базовый адрес). Напомним, что во время выполнения поддерживается указатель CEP (указатель текущей среды ссылок), так что в любой точке выполнения программы он указывает на базовый адрес записи активации в стеке подпрограммы, выполняющейся в данный момент. Если выполняется подпрограмма Sub и ссылается на переменную Y, то местоположение объекта данных, ассоциированного с Y, определяется посредством добавления смещения переменной Y к содержимому указателя CEP. Сам идентификатор Y также не требуется хранить в записи активации; для выполнения подпрограммы нужны только объекты данных. Такая реализация изображена на рис. 9.6. Пунктирная стрелка показывает смещение, вычисленное для ссылки на переменную Y.

Следуя общей модели реализации подпрограмм, описанной в разделе 8.3.2, достаточно легко реализовать оба подхода к моделированию среды ссылок — с удалением и сохранением локальных переменных. Несколько дополнительных моментов заслуживают внимания.

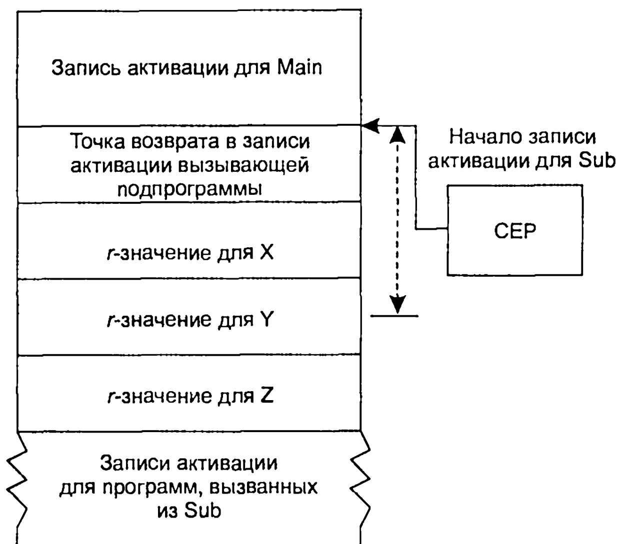


Рис. 9.6. Расположение удаляемых локальных переменных и ссылки на них

1. Для *простой* структуры вызова-возврата, описанной в начале этой главы, *сохранение* и *удаление* в отсутствие рекурсии приводит, по существу, к такой же самой реализации, поскольку не существует более одной записи активации, которая может быть размещена статически. Но если для переменных используются начальные значения, то возможны два различных подхода к реализации инициализации переменных (см. задачу 2).

2. *Отдельные переменные* с легкостью могут быть реализованы обоими способами — те переменные, которые требуется сохранить, располагаются в сегменте кода, а те, которые нужно уничтожить, размещаются в записи активации. Этот способ используется в PL/I: сохраняемые переменные объявляются с атрибутом STATIC, а удаляемые переменные объявляются как AUTOMATIC.

3. *Имя подпрограммы*, которое ассоциируется с объявлением подпрограммы в среде локальных ссылок, всегда может рассматриваться как сохраняемое. Ассоциация имени и определения может быть представлена указателем в сегменте кода, который указывает на сегмент кода, представляющий подпрограмму.

4. *Имя формального параметра* представляется объектом данных, который заново инициализируется новым значением при каждом вызове подпрограммы, как описано в разделе 9.3. Такая инициализация предотвращает сохранение старого значения формального параметра между вызовами подпрограммы. Таким образом, формальные параметры всегда трактуются как удаляемые ассоциации.

5. Если допускается *рекурсивный* вызов подпрограмм, то в один и тот же момент времени в центральном стеке может находиться несколько записей активации одной и той же подпрограммы Sub. Если переменная Y трактуется как подлежащая удалению, то в каждой записи активации должен содержаться свой

отдельный объект с именем Y, и во время выполнения каждой активации он будет ссылаться на свою собственную локальную копию объекта Y. Обычно желательно иметь отдельные копии объекта Y в каждой активации, поэтому в языках, допускающих рекурсивные подпрограммы (или иные структуры управления подпрограммами, приводящие к одновременному существованию в памяти нескольких активаций одной подпрограммы), как правило, используются удаляемые локальные среды. Тем не менее часто сохранение некоторых локальных переменных имеет большое значение. Например, в языке ALGOL 60 локальные переменные, объявленные с атрибутом own, становятся сохраняемыми переменными, хотя содержащая это объявление подпрограмма может быть рекурсивной. Если несколько активаций подпрограммы Sub ссылаются на сохраняемую переменную Y, то существует только один объект данных Y, который используется во всех активациях подпрограммы Sub, причем его значение сохраняется от одной активации до другой.

Преимущества и недостатки. Оба описанных подхода используются в большом количестве языков, в том числе и в наиболее важных языках. Подход с сохранением локальных данных позволяет писать подпрограммы, *чувствительные к предыстории*, поскольку результат каждого обращения к подпрограмме зависит не только от входных данных, но и от значений локальных данных, вычисленных во время предыдущих активаций. Подход с удалением локальных данных не позволяет сохранять их между вызовами, поэтому, если требуется сохранить между вызовами подпрограммы значение какой-либо переменной, она должна быть объявлена как нелокальная для данной подпрограммы. Для рекурсивных подпрограмм, однако, удаление локальных переменных является более естественной стратегией. Такой подход обладает также тем преимуществом, что в этом случае экономится место в памяти, так как требуется хранить таблицы локальных сред только тех подпрограмм, которые выполняются или выполнение которых приостановлено (но не завершено). Таблицы локальных сред для всех подпрограмм существуют на протяжении всего времени выполнения программы с использованием сохранения.

9.3. Передача параметров

Строго локальный объект данных используется в пределах одной среды локальных ссылок, то есть внутри одной подпрограммы. Но часто объекты данных используются совместно несколькими подпрограммами таким образом, что операции, определенные в каждой из подпрограмм, могут использовать эти данные. Объект данных может передаваться как явный параметр между подпрограммами, но возможны такие ситуации, когда использование явных параметров неудобно. Рассмотрим, например, множество подпрограмм, которое использует общую таблицу данных. Все подпрограммы должны иметь доступ к этой таблице, но передавать ее каждый раз при вызове очередной подпрограммы в виде явного параметра слишком утомительно. Такое совместное использование общих данных основывается на совместном использовании ассоциаций для идентификаторов. Если три подпрограммы P, Q и R должны иметь доступ к одной и той же переменной X, то подходящим решением будет просто позволить идентификатору X иметь одну и ту

же ассоциацию в каждой подпрограмме. Ассоциация для X становится частью локальной среды для одной из подпрограмм, и она же становится общей частью *нелокальной* среды двух других. Совместное использование объектов данных при помощи нелокальной среды является важной альтернативой совместному использованию данных через передачу параметров.

Явная передача параметров и результатов — основной альтернативный метод совместного использования данных подпрограммами. В отличие от использования сред нелокальных ссылок, где для решения этой задачи определенные нелокальные имена должны быть сделаны видимыми для подпрограммы, объекты данных, передаваемые как параметры и результаты, передаются без присоединения к ним имени. В получающей данные подпрограмме каждому объекту данных дается новое *локальное имя*, которое будет использоваться для ссылок на этот объект. Совместное использование данных через передачу параметров особенно полезно, когда подпрограмма при каждом вызове должна обрабатывать разные передаваемые ей данные. Например, если подпрограмма P используется таким образом, что при каждом очередном своем вызове она должна заносить новые данные в некоторую таблицу, совместно используемую несколькими подпрограммами, то, как правило, эта таблица реализуется при помощи ссылок на нелокальную среду, а заносимые в таблицу данные передаются как явные параметры при каждом вызове P.

В языках программирования используется четыре подхода к нелокальным средам:

1) явные общие среды ссылок;
2) неявные нелокальные среды, основанные на динамической области видимости;
3) на статической области видимости;
4) на наследовании.

В следующих подразделах мы обсудим передачу параметров и первые три из перечисленных подходов к нелокальным средам. Наследование мы уже рассматривали в главе 7.

9.3.1. Фактические и формальные параметры

Сначала мы рассмотрим совместное использование *данных* через параметры, а затем обсудим использование параметров для передачи *подпрограмм* и *меток операторов*. В предыдущих главах термин *аргумент* использовался для обозначения объекта данных (или некоторой величины), который пересылался подпрограмме или элементарной операции как один из ее операндов (то есть как фрагмент данных, используемый при вычислении). Термин *результат* обозначал фрагмент данных (объект данных или величину), который возвращался после выполнения подпрограммы или операции. Аргументы подпрограммы могут быть переданы ей или через параметры, или через нелокальные ссылки (и, реже, через внешние файлы). Аналогично, результаты работы подпрограммы могут быть возвращены через параметры, через присваивания значений нелокальным переменным (или файлам) или через явные значения функций. Таким образом, термины *аргумент* и *результат* относятся к данным, которые передаются подпрограмме и возвращаются из

нее посредством использования различных языковых механизмов. Если мы сконцентрируем внимание на параметрах и передаче параметров, то центральными для нас станут термины *фактический параметр* и *формальный параметр*.

Формальный параметр — это разновидность локального объекта данных в подпрограмме. В определении подпрограммы обычно перечисляются имена и объявления для формальных параметров как часть ее спецификации (заголовка). Имя формального параметра представляет собой простой идентификатор, а в его объявлении обычно задают тип и другие атрибуты, как и в случае объявлений обычных локальных переменных. Например, в заголовке процедуры на языке С

```
SUB(int X; char Y);
```

определяются два формальных параметра X и Y и объявляется, к какому типу они принадлежат. Тем не менее объявление формального параметра несколько отличается от объявления переменной. В зависимости от выбранного механизма передачи формальный параметр может быть псевдонимом фактического параметра или может просто содержать копию значения объекта данных, являющегося фактическим параметром.

Фактический параметр — это объект данных, совместно используемый вызывающей и вызываемой подпрограммами. Фактический параметр может быть локальным объектом данных, принадлежащим вызывающей подпрограмме, он может быть также формальным параметром самой вызывающей подпрограммы, он может быть нелокальным объектом данных, видимым из вызывающей подпрограммы, и, наконец, он может быть результатом вычисления функции, вызванной из вызывающей подпрограммы, который сразу же передается вызываемой подпрограмме. В точке вызова подпрограммы фактический параметр представляется выражением, называемым *выражением фактического параметра*, которое обычно выглядит так же, как и любое другое выражение в языке (например, как выражение, которое может появиться в операторе присваивания). Например, подпрограмма SUB, описанная выше, может быть вызвана с любым из следующих типов выражений фактического параметра.

Вызов процедуры в P	**Фактический параметр в P**
SUB(I,B)	I, B: локальные переменные в P
SUB(27,true)	27, true: константы
SUB(P1,P2)	P1, P2: формальные параметры в P
SUB(G1,G2)	G1, G2: глобальные или нелокальные переменные в P
SUB(A[I],D.B1)	Компоненты массивов и записей
SUB(I+3,FN(Q))	Результаты вычисления элементарных операций или определенных программистом функций

Синтаксис вызова процедуры в языке С является типичным для многих языков. Вызов подпрограммы, как сказано в разделе 8.2, записывается в префиксной форме, при этом сначала записывается имя подпрограммы, за которым в круглых скобках следует список выражений фактических параметров (хотя существуют и другие формы записи, например инфиксная запись в APL, кембриджская польская запись в LISP). Для простоты мы принимаем характерное для языка С пре-

фиксное представление и используем термины *список фактических параметров* и *список формальных параметров* для последовательности фактических и формальных параметров, указанных в вызове подпрограммы и в ее определении соответственно.

Если при вызове подпрограммы ей передается параметр в виде выражения фактического параметра любой из перечисленных выше форм, то во время вызова, но перед входом в подпрограмму *вычисляется* значение этого выражения. Объекты данных, полученные в результате вычисления выражений фактических параметров, становятся фактическими параметрами, переданными подпрограмме. Позже будет отдельно рассмотрен случай, когда выражения фактических параметров *не вычисляются* во время вызова подпрограммы, а передаются ей *не вычисленными*.

Установка соответствия

Когда подпрограмма вызывается со списком фактических параметров, следует установить соответствие между фактическими параметрами и формальными параметрами, перечисленными в определении подпрограммы. Мы последовательно опишем два метода, которые применяются для установления этого соответствия.

Позиционное соответствие. Соответствие между фактическими и формальными параметрами устанавливается на основе их позиций в списках соответственно фактических и формальных параметров: два параметра, которые занимают одинаковые позиции в списках, образуют пару. Таким образом, первый фактический и первый формальный параметры составляют пару, затем — вторые параметры из обоих списков и т. д.

Соответствие по имени. В языке Ada и в некоторых других языках при вызове подпрограммы можно явным образом указать, какой формальный параметр должен соответствовать данному фактическому параметру. Например, в языке Ada можно вызвать подпрограмму Sub следующим образом:

```
Sub(Y => B, X => 27);
```

Здесь при вызове подпрограммы Sub фактический параметр B объединяется в пару с формальным параметром Y, а фактический параметр 27 — с формальным параметром X.

В большинстве языков используется исключительно позиционное соответствие, поэтому в примерах, приведенных в нашей книге, также применяется этот метод. Обычно количество формальных и фактических параметров должно совпадать, чтобы соответствие между ними было взаимно однозначным. Тем не менее в некоторых языках это требование не выполняется, и в этом случае используются специальные соглашения, интерпретирующие отсутствие или избыток фактических параметров. В нашей книге для простоты мы всюду предполагаем, что количество параметров в обоих списках одинаково.

9.3.2. Методы передачи параметров

Когда подпрограмма передает управление другой подпрограмме, требуется связать фактические параметры вызывающей подпрограммы с формальными параметрами вызываемой подпрограммы. Чаще всего применяются два подхода: либо фактический параметр вычисляется и полученное значение передается формаль-

ному параметру, либо фактический объект данных передается формальному параметру. Мы описываем это как процесс, имеющий два этапа:

1) описание деталей реализации механизма передачи параметров;

2) описание семантики использования параметров.

Для передачи фактических параметров в подпрограмму было разработано несколько методов. Первые четыре из описываемых далее являются наиболее распространенными. Это передача по имени, передача по ссылке, передача по значению и передача по значению-результату.

Передача по имени. В этой модели передачи параметров вызов подпрограммы рассматривается как подстановка для всего тела подпрограммы. В такой интерпретации каждый формальный параметр заменяется фактическим значением каждого конкретного фактического параметра. В этом случае каждая ссылка на формальный параметр в теле подпрограммы требует вычисления заново значения соответствующего фактического параметра, как если бы были произведены действительные подстановки.

Эта модель требует, чтобы в точке вызова подпрограммы фактические параметры оставались не вычисленными до тех пор, пока в подпрограмме действительно не будет ссылки на них. Параметры передаются не вычисленными, и вызванная подпрограмма определяет сама, когда они действительно вычисляются, если вычисляются вообще. Напомним из приведенного ранее (глава 8) обсуждения унифицированных правил вычислений, что эта возможность оказалась полезна в трактовке операций типа условной операции `if-then-else` как обычных операций. Эта методика иногда бывает полезна для элементарных операций, но для подпрограмм, определяемых пользователем, ее полезность представляется проблематичной из-за большой стоимости реализации. Тем не менее передача параметров по имени играет важную роль в языке ALGOL, а также имеет большое теоретическое значение. Но этот метод приводит к значительным затратам при выполнении программ, поэтому его популярность невелика.

Основное правило передачи по имени может быть сформулировано в терминах подстановки: до начала выполнения подпрограммы везде в ее теле фактический параметр должен быть подставлен вместо формального параметра. Хотя это правило кажется на первый взгляд достаточно простым, рассмотрим проблему, возникающую уже в таком простом вызове подпрограммы `call Sub(X)`. Если в подпрограмме `Sub` формальным параметром является `Y`, то всюду в теле подпрограммы `Sub` его следует заменить на `X` до начала ее выполнения. Однако этого еще недостаточно, поскольку, когда во время выполнения подпрограмма `Sub` дойдет до ссылки на `X`, окажется, что ассоциация для `X`, на которую мы ссылаемся, является на самом деле ассоциацией в ***вызывающей подпрограмме***, а не в подпрограмме `Sub` (если такая ассоциация там существует). Когда `X` подставляется вместо `Y`, мы должны также указать другую среду ссылок при обработке ссылки на `X`. Также может возникнуть неоднозначность, если `X` уже является переменной, известной в подпрограмме `Sub`.

Не удивительно, что в базисном методе реализации передачи по имени фактические параметры приходится трактовать как простые подпрограммы без параметров, традиционно называемые «переходниками» (английский термин для таких подпрограмм «thunk»). Когда в подпрограмме встречается ссылка на формальный

параметр, соответствующий передаваемому по имени фактическому параметру, то выполняется подпрограмма-«переходник», скомпилированная для этого параметра. Результатом ее выполнения будет вычисление значения фактического параметра в соответствующей среде ссылок, которое и будет являться возвращаемым значением подпрограммы-«переходника».

Передача по ссылке. Передача по ссылке является, вероятно, наиболее распространенным механизмом передачи параметров. Передача объекта данных с помощью этого механизма означает, что подпрограмме становится доступным *указатель* на местоположение этого объекта (то есть его *l*-значение). *При этом расположение объекта данных в памяти не изменяется.* В начале выполнения подпрограммы *l*-значения всех фактических параметров используются для инициализации локальных областей памяти, отведенных под формальные параметры.

Передача параметров по ссылке осуществляется в два этапа.

1. В вызывающей подпрограмме вычисляется каждое выражение фактического параметра, чтобы получить указатели на объекты данных, соответствующие фактическим параметрам (то есть их *l*-значения). Список таких указателей сохраняется в общей области памяти, доступной также и для вызываемой подпрограммы (часто этой областью служит набор регистров или программный стек). Затем управление передается вызываемой подпрограмме в соответствии с алгоритмом, приведенным в предыдущей главе (то есть при необходимости создается запись активации для подпрограммы, устанавливается точка возврата и т. д.).
2. В вызванной подпрограмме список указателей на фактические параметры используется для получения требуемых *r*-значений этих параметров.

Во время выполнения подпрограммы ссылки на имена формальных параметров рассматриваются как ссылки на обычные локальные переменные (за исключением того, что может быть выполнена скрытая операция выбора по указателю). По окончании выполнения вызванной подпрограммы результаты возвращаются в вызывающую подпрограмму также через объекты данных, соответствующие фактическим параметрам.

Передача по значению. Если параметр *передается по значению* (в языке Ada 95 это называется *по копии*), то *r*-значение фактического параметра передается формальному параметру вызванной подпрограммы. Механизм реализации похож на модель передачи по ссылке, за исключением того, что:

1) когда вызывается подпрограмма, при передаче параметра по ссылке передается его *l*-значение, тогда как при передаче по значению передается *r*-значение;

2) при ссылке на параметр в подпрограмме в случае переданного по ссылке параметра для доступа к фактическому объекту данных используется *l*-значение, сохраняемое в формальном параметре, тогда как при передаче по значению формальный параметр содержит собственно значение этого объекта данных, которое и используется для вычислений.

Из этого обсуждения должно быть ясно, что при передаче по ссылке создается псевдоним фактического параметра, тогда как при передаче по значению мы не имеем такой ссылки. Если фактический параметр передан по значению, формаль-

ный параметр не имеет доступа к значению фактического параметра, чтобы изменить его. Любые изменения значения формального параметра, произошедшие во время выполнения подпрограммы, теряются, когда программа завершает свое выполнение.

Передача по значению-результату. Если параметр передается по значению-результату, формальный параметр является локальной переменной (объектом данных) того же типа, что и фактический параметр. В момент вызова подпрограммы значение фактического параметра (*r*-значение) копируется в объект данных, соответствующий формальному параметру, так что достигается такой же эффект, как если бы была выполнена операция явного присваивания значения фактического параметра формальному параметру. Во время выполнения подпрограммы каждая ссылка на имя формального параметра рассматривается как простая ссылка на локальную переменную, как при передаче параметра по значению. По завершении подпрограммы содержимое объекта данных, соответствующего формальному параметру, копируется в объект данных, соответствующий фактическому параметру, как если бы имело место явное присваивание значения формального параметра фактическому параметру. Таким образом, фактический параметр сохраняет свое значение вплоть до завершения подпрограммы, когда ему присваивается новое значение как результат выполнения подпрограммы.

Передача параметров по значению-результату появилась в языке ALGOL-W, который был разработан Никлаусом Виртом в 60-е гг. как преемник языка ALGOL, еще до того, как им был создан язык Pascal. ALGOL-W был реализован на компьютере IBM 360, на котором реализация передачи параметров по ссылке была относительно неэффективной, так как отсутствовала возможность получить доступ к *r*-значению фактического параметра посредством одной команды. На этом компьютере отсутствовала операция косвенного обращения к памяти. Передача параметров по значению-результату делала все параметры локальными переменными, непосредственно адресуемыми с помощью указателя текущей записи активации, что позволяло ускорить выполнение.

И передача по значению, и передача по значению-результату широко используются в реализациях языков программирования, и в большинстве случаев они дают одинаковый результат. Тем не менее все четыре описанных способа передачи параметров — по имени, по ссылке, по значению и по значению-результату — имеют отличия, из-за которых могут появляться одинаковые программы с различной семантикой, как мы покажем чуть позже.

Передача по значению-константе. Если параметр передается *по значению-константе*, то во время выполнения подпрограммы не допускается никакого изменения значения формального параметра (то есть не допускается ни присваивания нового значения, ни иной модификации значения параметра, а передать его в другую подпрограмму можно только по значению-константе). Формальный параметр, таким образом, во время выполнения подпрограммы рассматривается как локальная константа. Поскольку не допустимы никакие изменения его значения, возможны два варианта реализации этого способа передачи. Формальный параметр может рассматриваться в точности как параметр, переданный по значению, так что он является локальным объектом данных, начальное значение которого представляет собой копию значения фактического параметра. С другой стороны, он может

рассматриваться как параметр, переданный по ссылке, так что формальный параметр содержит указатель на объект данных, соответствующий фактическому параметру.

Передача параметра и по значению, и по значению-константе гарантирует, что фактический параметр не будет изменен в вызывающей подпрограмме. Таким образом, с точки зрения вызывающей подпрограммы фактический параметр является для нее только входным аргументом. Его значение не может быть модифицировано этой подпрограммой ни случайно, ни при попытке передать через него возвращаемый результат вычислений в вызывающую подпрограмму.

Передача по результату. Параметр, передаваемый *по результату*, используется только для передачи результатов вычисления назад в вызывающую подпрограмму. Начальное значение объекта данных, соответствующего фактическому параметру, не имеет никакого значения для подпрограммы и не может быть в ней использовано. Формальный параметр является локальной переменной (объектом данных), у которого начальное значение отсутствует (или инициализировано обычным способом, предусмотренным для локальных переменных). Когда выполнение подпрограммы завершается, конечное значение формального параметра присваивается фактическому параметру, как в случае вызова по значению-результату.

В большинстве языков программирования используется один или два механизма передачи параметров. В FORTRAN применяется передача по ссылке, в то время как в Pascal используются два механизма передачи: по ссылке и по значению. Вызов по ссылке обозначается как `var X : integer`, а вызов по значению — как `X : integer`. (Следует отметить, что здесь кроется источник многочисленных ошибок при написании программ на Pascal. Если программист забудет включить ключевое слово `var`, параметр будет передан по значению, и любые изменения его значения не будут отражены при возврате в вызывающую подпрограмму. Эта ошибка очень коварна, так как бывает чрезвычайно трудно ее обнаружить.) В отличие от Pascal в языке С реализуется только вызов по значению. Однако использование указателей позволяет создавать параметры, передаваемые по ссылке. Передача по ссылке аргумента `i` процедуре `mysubroutine` осуществляется через выражение `&i`, которое передает *l*-значение аргумента `i`. Вызов процедуры будет выглядеть следующим образом: `mysubroutine(&i)`. При этом объявление самой процедуры выглядит как `mysubroutine(int *x)` — это означает, что `x` является указателем на целочисленный объект данных. Если забыть включить подходящие операции разыменования (`&` и `*`), то это приведет ко многим ошибкам при программировании на С.

9.3.3. Семантика передачи параметров

Обсуждение способов передачи параметров, приведенное в предыдущей главе, вынуждает программиста составить представление о фактической реализации этого процесса, прежде чем выбрать какой-либо режим передачи параметров. Но это на самом деле противоречит одному из принципов разработки языков программирования: программист не должен беспокоиться о деталях реализации языка. В языке Ada, например, используется другой подход. Вместо того чтобы явно указывать способ передачи, программист должен только определить *роль* данного параметра.

Параметр может быть определен с помощью ключевого слова in как входной параметр, что означает передачу значения фактического параметра формальному параметру с последующим его использованием в подпрограмме. Параметр может быть определен с помощью ключевого слова out как выходной параметр, значение которого генерируется в вызванной подпрограмме и затем передается назад фактическому параметру при выходе из подпрограммы. И наконец, параметр может быть определен с помощью ключевых слов in out одновременно как входной и выходной. В этом случае его значение передается в вызванную подпрограмму, а затем значение результата передается обратно фактическому параметру. Язык Ada определен таким образом, что при его реализации для некоторых из перечисленных типов параметров можно выбрать альтернативные методы их передачи. Для программиста выбор того или иного метода реализации не оказывает влияния на конечный результат, если только вызванная программа нормально завершается и не имеет доступа к фактическим параметрам через псевдонимы.

В исходном определении языка Ada (Ada 83) разработчик имел возможность выбора между передачей по ссылке и передачей по значению для реализации входных (in) и выходных (out) параметров. Это приводило к проблемам согласования, так как *правильные* программы могли давать различные результаты при компиляции с помощью различных *правильных* компиляторов. Поэтому в пересмотренном варианте языка Ada 95 подход к передаче параметров был пересмотрен.

- *Элементарные типы данных* (например, скалярные величины типа целых чисел, вещественных чисел или булевых значений) передаются по значению-константе, если они являются входными (in) параметрами, и по значению-результату, если являются выходными (out) или одновременно входными и выходными (in out) параметрами.
- *Составные типы данных* (например, массивы и записи) передаются по ссылке.

Эти нововведения упростили передачу параметров в Ada, но все же некоторая возможность путаницы при выполнении подпрограмм остается (см. задачу 23 в конце этой главы).

Явные значения функций

В большинстве языков, если результатом подпрограммы является только одно значение, его можно возвратить как явное *значение функции*, а не через параметр. В этом случае подпрограмма должна быть объявлена как подпрограмма-функция, а тип возвращаемого результата должен быть задан в спецификации этой подпрограммы, как, например, в объявлении C `float fn(int a)`, которое определяет fn как подпрограмму-функцию, возвращающую результат вещественного типа. В теле подпрограммы-функции результат, возвращаемый как значение функции, может быть определен одним из двух способов. В первом способе, используемом в языке C, возвращаемое значение функции следует задать в виде явного выражения в операторе return, который завершает выполнение подпрограммы (например, `return 2 * x` для указания того, что значение выражения `2 * x` должно быть возвращено как результат выполнения подпрограммы-функции). В Pascal используется альтернативный способ: возвращаемое значение присваивается имени функции внутри

подпрограммы-функции (например, fn := 2 * x). В этом способе подпрограмма может содержать несколько операторов присваивания значения имени функции. При этом в вызывающую подпрограмму возвращается последнее из присвоенных перед завершением подпрограммы значений. В обоих способах, возможно, было бы лучше рассматривать возвращаемое значение как дополнительный неявный выходной (out) параметр подпрограммы.

9.3.4. Реализация передачи параметров

Поскольку каждая активация подпрограммы получает разный набор значений параметров, память под формальные параметры подпрограммы обычно выделяется в записи активации подпрограммы, а не в сегменте кода. Каждый формальный параметр является в подпрограмме локальным объектом данных. Если в заголовке подпрограммы указано, что формальный параметр P принадлежит некоторому типу T (то есть фактический параметр, соответствующий данному формальному, является объектом данных типа T), то формальный параметр реализуется одним из двух способов, в зависимости от используемого способа передачи (как обсуждалось выше). Формальный параметр P рассматривается либо как локальный объект данных типа T (начальное значение которого может быть копией значения фактического параметра), либо как локальный объект данных типа *указатель* на объект данных типа T (начальным значением которого является указатель на объект данных, соответствующий фактическому параметру). Первый из этих двух методов используется для передачи параметров по значению-результату, по значению и по результату; второй используется для передачи параметров по ссылке. Любой из этих методов можно использовать для реализации передачи параметров по значению-константе. Явное значение функции можно трактовать как передачу параметра с помощью первого из описанных методов. Если в определении языка не предусмотрена спецификация типов для формальных параметров (как, например, в LISP, APL и SNOBOL4), то формальный параметр может быть реализован как локальная переменная-указатель, но хранящийся в ней указатель может указывать на объект данных произвольного типа.

Различные действия, связанные с передачей параметров, можно разбить на две группы: те, которые связаны с *точкой вызова* подпрограммы в каждой вызывающей подпрограмме, и те, которые связаны *с входом и выходом* из подпрограммы. В точке вызова в каждой вызывающей подпрограмме вычисляется каждое выражение фактического параметра и создается список указателей (а иногда — просто копии их значений) на объекты данных, соответствующие фактическим параметрам. Важно понимать, что это вычисление происходит в точке вызова, в среде ссылок вызывающей подпрограммы. Когда фактические параметры определены, управление передается вызываемой подпрограмме. Обычно это влечет за собой изменения в значениях указателей CEP и CIP (как описано в главе 8), что приводит к передаче управления на начало выполняемого кода вызванной подпрограммы и также изменению среды ссылок на среду соответствующей вызванной подпрограммы.

После передачи управления подпрограмме в ее *прологе* завершаются действия, связанные с передачей параметров, — в формальный параметр копируется либо

значение фактического параметра, либо указатель на фактический параметр. Перед завершением подпрограммы ее *эпилог* должен скопировать возвращаемые результаты в фактические параметры, переданные по результату или по значению-результату. Значения функций также должны быть скопированы в регистры или во временную память, предоставленную вызывающей подпрограммой. Затем подпрограмма заканчивается и ее запись активации уничтожается, так что обычно все результаты должны быть скопированы из записи активации еще до завершения подпрограммы.

При реализации передачи параметров компилятор выполняет две основные задачи. Сначала он должен сгенерировать правильный выполняемый код для передачи параметров, возвращения результатов и для всех ссылок на имена формальных параметров. Поскольку в большинстве языков предусмотрено несколько методов передачи параметров, необходимый в каждом случае выполняемый код часто слегка различается. Генерация этого кода также является непростой задачей, поскольку затрагивает координацию действий, выполняемых в каждой точке вызова подпрограммы, в ее прологе и эпилоге. Второй важной задачей компилятора является выполнение необходимой статической проверки типов для гарантии того, что тип каждого объекта данных, соответствующего фактическому параметру, совпадает с типом, объявленным для соответствующего формального параметра. Для выполнения этой проверки компилятор должен знать спецификацию вызываемой подпрограммы (количество, порядок и типы параметров), а внутренняя структура тела подпрограммы не имеет значения. Эта спецификация должна быть доступна компилятору в любой точке вызова подпрограммы. Во многих языках, особенно в тех, в которых допускается раздельная компиляция подпрограмм, если подпрограмма `Q` вызывается из подпрограммы `P`, то при компиляции `P` должна быть предоставлена отдельная спецификация `Q`, даже если определение подпрограммы `Q` находится в каком-либо другом месте. Это необходимо для того, чтобы компилятор мог выполнить статическую проверку типов и сгенерировать соответствующий код для передачи параметров в каждой точке вызова.

Примеры передачи параметров

Комбинация методов передачи параметров с различными типами фактических параметров приводит к разнообразным эффектам. Некоторые примеры использованы для объяснения различных тонкостей передачи параметров. В приведенных примерах используются два метода передачи параметров: *по ссылке* и *по значению*. Отличия, возникающие при использовании других методов, рассматриваются в задачах в конце главы. Все примеры написаны на языке С. В нашей версии С имя формального параметра с предшествующим символом «звездочка» (*) передается по ссылке, а при ее отсутствии — по значению.

Простые переменные и константы. В листинге 9.7 (*а*) приведена подпрограмма `Q` на языке С с двумя формальными параметрами: `i`, передаваемым по значению, и `j`, передаваемым по ссылке. Предположим, что мы написали подпрограмму `P`, вызывающую эту подпрограмму `Q` с двумя целочисленными переменными `a` и `b` в качестве фактических параметров (листинг 9.7, *б*). При выполнении подпрограммы `P` два оператора печати выведут следующие числа: `12 13 2 13`. Проследим за каждым из параметров по очереди.

Когда P вызывает Q, вычисляются выражения фактических параметров a и &b (то есть вызывается операция обработки ссылок для определения текущей ассоциации для имен a и b). Каждое имя представляет объект данных для целых переменных, так что переданные фактические параметры — это *r*-значение переменной a и *l*-значение переменной b. Поскольку a передается по значению, формальный параметр i представлен локальной целой переменной внутри подпрограммы Q. Когда подпрограмма Q начинает выполняться, значение переменной a в момент вызова Q присваивается формальному параметру i как его начальное значение. Далее между a и i какая-либо связь отсутствует. Поэтому, когда i присваивается новое значение 12, то a не меняется. После завершения вызова Q значение переменной a по-прежнему равно 2.

В отличие от a фактический параметр b передается по ссылке. Это означает, что формальный параметр j представляется локальной переменной подпрограммы Q, типом которой является *указатель на целочисленный объект данных*. Когда Q начинает выполняться, указатель на объект данных b сохраняется как *r*-значение формального параметра j. Когда к значению формального параметра j добавляется 10, j не меняется. Вместо этого каждая ссылка на j (*r*-значение j, которое является *l*-значением b) после выполнения операции выборки в соответствии с заданным значением указателя в действительности получает доступ к местоположению объекта данных b. В итоге операция присваивания переменной j, хотя и выглядит так же, как присваивание нового значения переменной i, имеет совершенно иной смысл. Значение фактического параметра b изменилось, теперь оно равно 13. При выводе на печать значений параметров i и j в подпрограмме Q получается соответственно 12 и 13. После возвращения в подпрограмму P значения соответствующих фактических параметров a и b снова выводятся на печать. Видно, что изменилось только значение переменной b. Значение 12, присвоенное i в подпрограмме Q, теряется по ее завершении, поскольку локальные переменные уничтожаются после окончания работы подпрограммы. Значение j, конечно, тоже утрачивается, но это значение является указателем на объект данных, а не числом 13.

Листинг 9.7. Передача параметров по ссылке и по значению в языке C: *а* — вызванная подпрограмма, *б* — вызывающая подпрограмма

```
а)
Q(int i, int *j)
   { i=i+10;
   *j=*j+10;
   printf("i=%d; j=%d;\n",i,*j);
   }
б)
P()
   {int a, b;
   a = 2;
   b = 3;
   Q(a,&b);
   printf("%d %d\n",a,b);
   }
```

Структуры данных. Предположим, что мы хотим написать другую версию подпрограммы Q, в которой формальными параметрами являются векторы. На C та-

кую подпрограмму написать сложно, поскольку обычно в С значения массивов не передаются по значению. Объявление процедуры `sub1(int a[20])` в С интерпретируется так же, как если бы было записано `sub1(int *a)`, то есть в С передача массива осуществляется так же, как в Ada 95, передачей не его *r*-значения (копии массива), а указателя на массив (его *l*-значения). По этой причине данный пример мы приводим на языке Pascal, который скопирует *r*-значение массива-параметра, передаваемого по значению в подпрограмму:

```
type vect = array [1..3] of integer;
procedure Q(k: vect; var l: vect);
   var n: integer;
   begin
      k[2] := k[2] + 10;
      l[2] := l[2] + 10;
      for n := 1 to 3 do write(k[n]);
      for n := 1 to 3 do write(l[n])
   end;
```

Процедуру P можно записать так, как это сделано в листинге 9.8.

После ее выполнения будут напечатаны следующие значения: 6 17 8 6 17 8 6 7 8 6 17 8.

Листинг 9.8. Структуры данных как параметры подпрограммы

```
procedure P;
   var c,d: vect;
      m: integer;
   begin
      c[1] := 6; c[2] := 7; c[3] := 8;
      d[1] := 6; d[2] := 7; d[3] := 8;
      Q(c,d);
      for m := 1 to 3 do write(c[m]);
      for m := 1 to 3 do write(d[m])
   end;
```

Чтобы проследить передачу параметров c и d, сначала следует заметить, что вычисление выражений фактических параметров c и d в P приводит к указателям на блоки памяти для векторов c и d в записи активации подпрограммы P (как в предыдущем примере для параметров a и b). Эти указатели передаются подпрограмме Q. Поскольку вектор c передается по значению, соответствующий формальный параметр k является локальным массивом в подпрограмме Q, имеющим такой же вид, что и c (три компонента плюс дескриптор). Три компонента вектора c копируются в соответствующие компоненты вектора k, после чего c и k более не взаимодействуют. Поэтому, когда управление возвращается к подпрограмме P, вектор c остается неизменным, даже если в подпрограмме Q происходили присваивания новых значений компонентам вектора k. Но вектор d, который передается по ссылке, изменяется в результате присваивания новых значений компонентам вектора l, являющегося формальным параметром, так как l — это не вектор, а лишь указатель на вектор. Когда начинает выполняться подпрограмма Q, формальный параметр l инициализируется указателем на вектор d, и каждая последующая ссылка на него в Q приводит через этот указатель к d. Таким образом, присваивание нового значения компоненту l[2] также изменяет и компонент d[2]. Печатаемые значения отражают эти отличия. На рис. 9.7 изображен стек времени выполнения перед завершением подпрограммы Q.

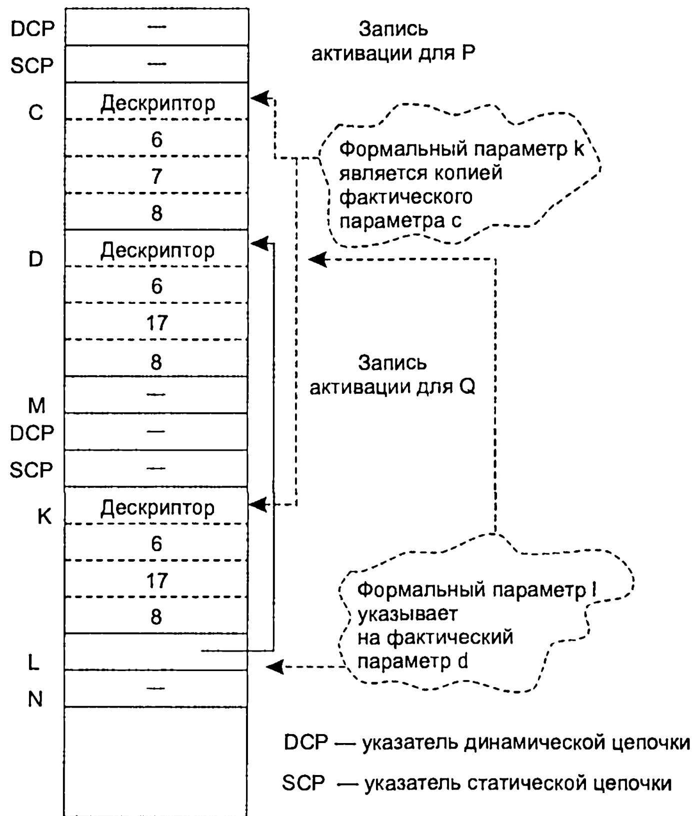


Рис. 9.7. Стек времени выполнения перед завершением подпрограммы Q

Вообще говоря, такая структура данных, как массив или запись, передаваемая по значению, обычно копируется в соответствующую структуру данных (формальный параметр) в вызванной подпрограмме, которая работает с этой локальной копией и не имеет доступа к оригиналу. Структура данных, переданная по ссылке, не копируется, а вызванная подпрограмма работает непосредственно со структурой данных, представляющей фактический параметр (используя для доступа указатель, хранящийся в формальном параметре).

Компоненты структур данных. Вернемся к процедуре Q из листинга 9.7 (*a*), но теперь вместо простых переменных или констант передадим Q в качестве параметров компоненты каких-либо структур данных — например, написав подпрограмму P следующего вида:

```
P()
   {int c[4];
   int m;
   c[1] = 6; c[2] = 7; c[3] = 8;
   Q(c[1],&c[2]);
   for (m = 1; m<=3; m++) printf("%d\n",c[m]);
}
```

Когда P выполнится, будут напечатаны значения: 16 17 6 17 8.

Передача c[1] по значению происходит по тому же сценарию, что и прежде. Выражение c[1] вычисляется получением ссылки на вектор c и последующим выбором первого компонента. Результатом будет *r*-значение этого компонента. Формальный параметр i инициализируется, как и прежде, этим вычисленным *r*-значением, поэтому и остальные действия остаются такими же, как и в случае передачи переменных. Подобным же образом вычисляется &c[2], и в подпрограмму Q передается указатель на компонент вектора. Тогда операции присваивания в подпрограмме Q непосредственно изменяют компонент вектора c через указатель, хранящийся в формальном параметре j.

Компоненты вектора c представлены *в памяти точно так же, как были бы представлены объекты данных простых переменных того же типа.* Выполняемый код подпрограммы Q, который манипулирует формальными параметрами i и j, будет одним и тем же как при вызове Q(a, b), так и при вызове Q(c[1], c[2]). Если бы компоненты массива c были представлены в памяти каким-либо образом, отличным от представления простых переменных a и b (например, были бы каким-то образом упакованы), то до вызова подпрограммы Q потребовалось бы преобразовать фактические параметры к нужному виду (который ожидается в Q), разместить эти преобразованные параметры во временной памяти и передать указатели на расположение во временной памяти этих параметров в подпрограмму Q. При передаче по значению этих преобразований было бы вполне достаточно, но при передаче по ссылке передаваемый указатель больше не является указателем на исходный объект данных, поэтому присваивания в подпрограмме Q больше не модифицируют непосредственно фактический параметр. По этой причине передача компонентов упакованных массивов и записей часто запрещается (как, например, в Pascal).

Компоненты массивов с вычисляемыми индексами. Предположим, что в подпрограмме R имеются два целочисленных параметра, передаваемых по ссылке:

```
R(int *i,int *j)
   {*i = *i + 1;
   *j = *j + 1;
   printf("%d %d\n",*i,*j);
   }
```

Если в подпрограмме P вместо вызова Q подставить вызов R, то получится:

```
P()
   {int c[4];
   int m;
   c[1] = 6; c[2] = 7 ; c[3] = 8;
   m = 2;
   R(&m, &c[m]);
   for (m = 1; m<=3; m++) printf("%d\n",c[m]);
   }
```

Какие значения теперь будут распечатаны при выполнении P? Отметим, что начальное значение m равно 2; но поскольку этот параметр передается по ссылке, то при выполнении R его значение поменялось на 3, прежде чем c[m] увеличилось на 1 через указатель, хранящийся в j. Но тогда добавит ли оператор *j = *j + 1 единицу к c[2] или к c[3]? Очевидно, что это будет компонент c[2], а не c[3], так как выражение фактического параметра c[m], определяющее значение указателя на компонент массива c, вычисляется в точке вызова подпрограммы R. Но во время

вызова R значение m было равно 2, поэтому подпрограмме R передается указатель на второй элемент массива c. Более того, подпрограмме R вообще ничего не известно о существовании c[3], так как внутри R указатель на c[2] воспринимается так же, как указатель на любой объект данных простой целой переменной. Таким образом, будут напечатаны следующие значения: 3 8 6 8 8. (Если бы параметры передавались *по имени*, был бы другой эффект.)

Указатели. Предположим, что фактический параметр является простой переменной типа *указатель* или структурой данных типа массива или записи, компонентами которой являются *l*-значения (указатели на объекты данных). Например, предположим, что в подпрограмме P имя x объявлено как vect *x и соответствующий формальный параметр в Q объявлен аналогичным образом: vect *h. Независимо от того, объявлен ли параметр h как передаваемый по ссылке или по значению, эффект от передачи x в качестве фактического параметра заключается в том, что подпрограмма Q получает прямой доступ к вектору, на который указывает x. Если передача осуществляется по значению, то Q имеет собственную копию *l*-значения, которая содержит x, так что и x, и h указывают на один и тот же вектор. Если передача происходит по ссылке, то h содержит указатель на x, который, в свою очередь, содержит указатель на вектор. Можно сформулировать общее правило: если фактический параметр содержит указатель или компоненты, содержащие указатели, то объекты данных, на которые ссылаются эти указатели, непосредственно доступны из вызываемой подпрограммы, независимо от метода передачи параметров. Отметим, что если в качестве параметра подпрограмме передается *по значению* связанный список (или какая-нибудь другая связанная структура данных), то обычно это означает, что в подпрограмму во время передачи копируется только указатель на первый элемент; вся связанная структура не копируется. Этим свойством переменных-указателей и добавлением различных операций с *r*- и *l*-значениями в C объясняется то, что в языке C не требуется наличия явного механизма передачи параметров по ссылке, а достаточно передачи по значению.

Результаты выражений. Если мы хотим передать по ссылке какое-либо выражение, например a + b, в подпрограмму Q(&(a + b),&b), то транслятор должен вычислить значение выражения в точке вызова, записать это значение *во временную область памяти* в P и передать указатель на эту область в подпрограмму Q как параметр. Подпрограмма Q выполняется так же, как и раньше. Передача по ссылке приводит к тому, что формальный параметр содержит указатель на временную область памяти в P. Поскольку эта область не имеет никакого имени, по которому на нее можно было бы ссылаться в P, то любые присваивания ей новых значений в подпрограмме Q не изменяют значение, на которое P впоследствии может сослаться. Таким образом, подпрограмма Q не может вернуть результат в P через параметр, передаваемый по ссылке. И в Pascal, и в C этот случай, когда передача является передачей по ссылке, запрещен, поскольку присваивания формальному параметру не вызывают видимого эффекта в вызывающей подпрограмме (и поэтому в таких случаях следует использовать передачу по значению).

Псевдонимы и параметры

Возможность использования псевдонимов (нескольких имен для одного и того же объекта данных в одной среде ссылок) в большинстве языков программирования

возникает в связи с передачей параметров. Как было показано в разделе 9.2.1, использование псевдонимов вызывает проблемы в понимании и проверке правильности программ, а также затрудняет оптимизацию. Псевдоним может быть создан в процессе передачи параметров одним из двух способов.

1. *Формальный параметр и нелокальная переменная.* Подпрограмма может иметь прямой доступ к объекту данных, переданному как фактический параметр по ссылке, через нелокальное имя. Тогда имя формального параметра и нелокальное имя становятся псевдонимами, так как каждое из них ссылается на один и тот же объект данных. В листинге 9.2 приведен пример такого способа создания псевдонимов.

2. *Два формальных параметра.* Один и тот же объект данных может быть передан в качестве фактического параметра по ссылке в двух позициях одного и того же списка фактических параметров. Тогда два имени соответствующих формальных параметров становятся псевдонимами, так как любое из них может быть использовано для ссылки на исходный объект данных. Например, процедура R, определенная в языке Pascal при помощи следующей спецификации:

   ```
   procedure R(var i,j: integer);
   ```

   может быть вызвана из P таким образом: R(m,m). Во время выполнения R и переменная i, и переменная j содержат указатели на один и тот же объект данных m в P, поэтому i и j являются псевдонимами. В языке FORTRAN такой способ создания псевдонимов запрещен.

Подпрограммы как параметры

Во многих языках подпрограмма может быть передана другой подпрограмме в качестве фактического параметра. В этом случае выражение фактического параметра состоит из *имени* передаваемой подпрограммы. Тогда соответствующий формальный параметр определяется как параметр, тип которого — *подпрограмма.* Например, в языке Pascal можно определить подпрограмму Q с формальным параметром R, тип которого — процедура или функция:

```
procedure Q(x: integer; function R(y,z: integer): integer);
```

Тогда Q можно вызывать, передавая ей в качестве второго параметра подпрограмму-функцию (например, вызов Q(27, fn) вызывает подпрограмму Q и передает подпрограмму-функцию fn в качестве параметра). Внутри Q подпрограмма, переданная как параметр, может быть вызвана с использованием имени формального параметра R. Например, z := R(i, x) вызывает подпрограмму, являющуюся фактическим параметром (функция fn в предшествующем вызове), в которую передаются фактические параметры i и x. Таким образом, в данном случае вызов R(i, x) эквивалентен вызову fn(i, x). Если при следующем обращении к подпрограмме Q ей будет передан другой фактический параметр-подпрограмма, например функция fn2, то R(i, x) будет эквивалентно fn2(i, x).

С передачей подпрограмм в качестве параметров связаны две основные проблемы.

Статическая проверка типов. Когда подпрограмма, переданная как параметр в другую подпрограмму, вызывается с использованием имени формального пара-

метра (например, R(i, x)), важно, чтобы существовала возможность статической проверки типов, гарантирующая, что этот вызов включает правильное количество фактических параметров, а их типы соответствуют типам формальных параметров вызываемой подпрограммы. Поскольку фактическое имя вызываемой через формальный параметр подпрограммы не известно в точке вызова (в нашем примере это fn при первом вызове и fn2 при втором), то обычно компилятор не может без дополнительной информации определить, соответствуют ли фактические параметры i и x в обращении R(i, x) тем, которые ожидаются подпрограммой fn или fn2. Компилятору требуется полная спецификация формального параметра R, где указан не только тип *процедура* или *функция*, но также количество, порядок и тип каждого параметра (и результата) этой процедуры или функции (как, например, в спецификации подпрограммы Q). Тогда в пределах подпрограммы Q каждый вызов R может быть статически проверен на правильность его списка параметров. Помимо этого для каждого вызова Q можно проверить, совпадает ли спецификация фактического параметра, соответствующего R, спецификации, заданной для формального параметра R. Таким образом, и подпрограмма fn, и подпрограмма fn2 должны иметь одинаковое количество, порядок и типы параметров в соответствии со спецификацией для формального параметра R.

Нелокальные ссылки (свободные переменные). Предположим, что подпрограмма, такая как fn или fn2, содержит ссылку на нелокальную переменную. Например, предположим, что в fn присутствует ссылка на z, а в fn2 — на z2, и ни в одной из подпрограмм нет локального определения для переменной, на которую они ссылаются. Такая нелокальная ссылка часто называется *свободной переменной*, так как она не имеет локального связывания в пределах определения подпрограммы. Обычно, когда вызывается подпрограмма, формируется среда нелокальных ссылок, и эта среда используется во время выполнения подпрограммы в качестве средства для определения значения каждой ссылки на нелокальную переменную (как описано в следующих разделах). Но предположим, что подпрограмма fn, в которой содержится нелокальная ссылка, передается в качестве параметра из вызывающей подпрограммы P в вызванную подпрограмму Q. Какая среда нелокальных ссылок должна использоваться, когда fn вызывается в Q (при использовании соответствующего формального параметра R, например R(i, x) для вызова fn(i, x))? На первый взгляд, самый простой ответ заключается в том, что среда нелокальных ссылок должна быть такой же, как если бы в подпрограмме Q вызов R(i,x) был просто заменен вызовом fn(i,x), но оказывается, что в большинстве случаев этот ответ будет неверным. В листинге 9.9 приведена программа, иллюстрирующая возникающую трудность. Функция fn содержит нелокальные ссылки на x и на i. Согласно правилам определения статической области видимости языка Pascal, идентификатор x ссылается на переменную x, объявленную в главной программе, а i — на переменную i, объявленную в процедуре P. Тем не менее P передает fn в качестве параметра процедуре Q, и именно эта процедура в действительности вызывает fn через имя формального параметра R. В Q имеются локальные определения как для i, так и для x, и нельзя позволить fn использовать эти неправильные локальные переменные при ее вызове.

Проблема свободных переменных в функциях, передаваемых подпрограммам в качестве параметров, возникает не только в языках с блочной структурой и пра-

вилами определения статической области видимости, подобных языку Pascal. Она также встречается в языке LISP и других языках, которые используют для определения среды нелокальных ссылок правило последней ассоциации. Общее решение проблемы заключается в использовании следующего правила определения значения свободных переменных в параметрах-функциях: нелокальная ссылка (ссылка на свободную переменную) во время выполнения подпрограммы, переданной в качестве параметра, должна означать то же самое, что она означала бы, если бы подпрограмма была вызвана в точке, где она появляется в качестве фактического параметра в списке параметров. Например, в листинге 9.9 подпрограмма fn появляется как фактический параметр в списке параметров при вызове Q из P. Таким образом, нелокальные ссылки на x и i в fn независимо от того, где fn будет фактически вызвана впоследствии (в нашем случае внутри Q), должны означать то же, что они означали бы, если бы fn была вызвана в том же месте, где вызвана Q внутри P.

Листинг 9.9. Свободные переменные как параметры подпрограммы в языке Pascal

```
program Main;
   var x: integer;
   procedure Q(var i:integer; function R(j:integer):integer);
      var x:integer;
      begin
      x := 4;
      write("In Q, before call of R, I=",i,"X=",x);
      i := R(i);
      write("In Q, after call of R, I=",i,"X=",x)
      end;
   procedure P;
      var i: integer;
      function fn(k:integer):integer;          | B fn,
         begin;                                | I и X здесь
         x := x+k;                             | являются
         fn := i+k;                            | свободными
         write("In P, I=",i,"K=",k,"X=",x)     | переменными
         end;                                  |
      begin
      i := 2;
      Q(x,fn);
      write("In P, I=",i,"X=",x)
      end;
   begin
   x := 7;
   P;
   write("In Main, X=",x)
   end.
```

Для правильной реализации этого правила, определяющего значения нелокальных ссылок в подпрограммах, передаваемых в качестве параметров, должна существовать возможность воссоздания среды нелокальных ссылок в точке вызова такой подпрограммы через параметр, чтобы при ее выполнении использовалась правильная среда ссылок. Это достаточно просто сделать в случае реализации разрешения нелокальной ссылки с помощью метода статической цепи

(см. раздел 9.4.2). Необходимо лишь определить правильный указатель на параметр-подпрограмму в статической цепи и передать его в составе информации, передаваемой вместе с параметром-подпрограммой. Тогда параметр-подпрограмма превращается в пару указателей (CP, SCP), где CP — указатель на сегмент кода подпрограммы, а SCP — указатель в статической цепи, который используется, когда вызывается подпрограмма. Эта пара может передаваться через многочисленные уровни вызовов подпрограмм, пока не настанет момент вызова подпрограммы. В этой точке создается запись активации, устанавливается переданный SCP и начинается выполнение сегмента кода подпрограммы, как при любом другом вызове.

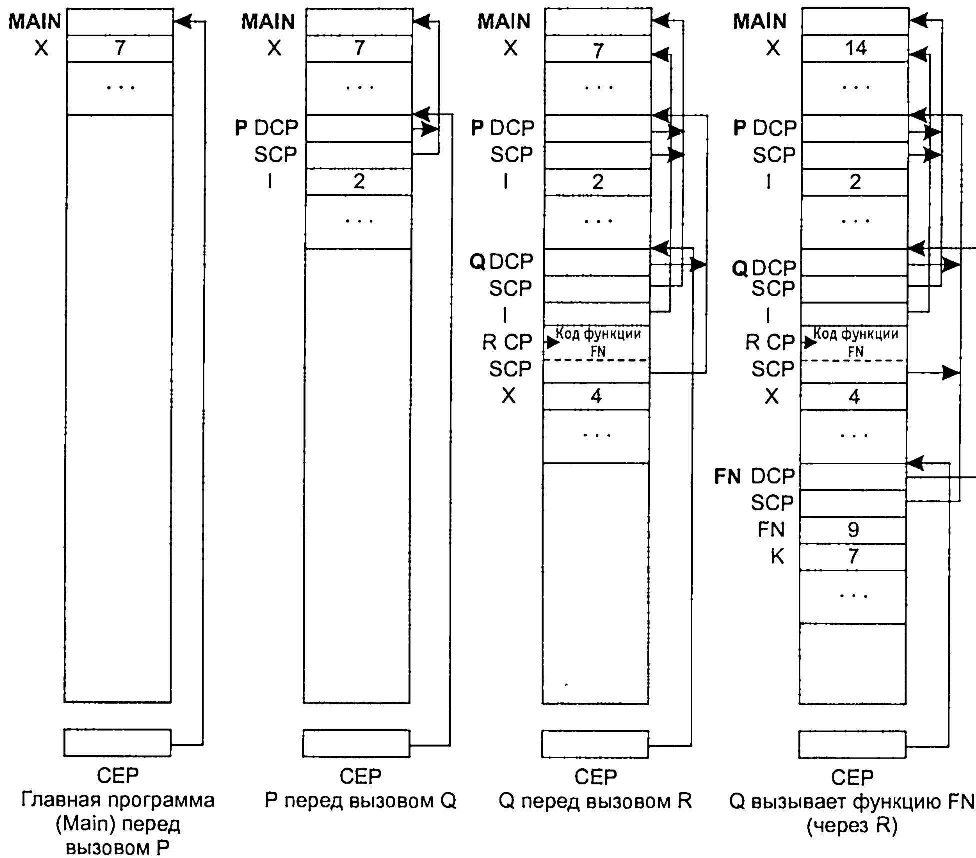


Рис. 9.8. Состояния центрального стека во время выполнения Pascal-программы

На рис. 9.8 проиллюстрированы основные этапы выполнения программы на Pascal, приведенной в листинге 9.9. Для того чтобы продемонстрировать взаимодействие статической (SCP) и динамической (DSP) цепей, а также среды нелокальных ссылок и параметров подпрограммы, в этом примере приведены значения переменных, отмечены точки возврата и указатели в статической цепи для основных

этапов выполнения программы. Представленные ниже результаты выполнения операторов печати дают представление о поведении программы:

В процедуре Q:	Перед вызовом R, I=7, X=4
В процедуре FN:	I=2, K=7, X=14
В процедуре Q:	После вызова R, I=9, X=4
В процедуре P:	I=2, X=9
В процедуре Main:	X=9

Метки операторов как параметры

Во многих языках допускается передавать подпрограммам в качестве параметров метки операторов, которые затем могут быть использованы как объекты оператора goto внутри подпрограммы. Помимо обычных сложностей, связанных с использованием goto, которые мы обсуждали в главе 8, этот механизм создает дополнительно две новые проблемы.

Какая активация должна использоваться? Во время выполнения подпрограммы метка оператора ссылается на определенную команду в сегменте ее кода. Но оператор goto не может просто передать управление этой команде, изменив CIP обычным способом, поскольку упомянутый сегмент кода может совместно использоваться несколькими активациями данной подпрограммы. Поэтому метка оператора, переданная в качестве параметра, должна указывать команду в *определенной активации* подпрограммы. Таким образом, метка становится парой значений (указатель команды, указатель записи активации), передаваемых подпрограмме как единый параметр.

Как реализуется оператор goto *перехода на метку, являющуюся параметром подпрограммы?* Когда выполняется оператор goto, объект перехода которого задан формальным параметром с объявленным типом «метка», то в большинстве случаев недостаточно просто передать управление указанной команде в указанной активации подпрограммы. Вместо этого обычно приходится просматривать всю динамическую цепь вызовов подпрограмм, пока не будет достигнута искомая запись активации. Это значит, что текущая подпрограмма, в которой выполняется оператор goto, должна быть завершена, то же касается и подпрограммы, вызвавшей данную и т. д., пока не будет достигнута активация подпрограммы, на которую указывает параметр-метка оператора goto. Затем начинает выполняться эта активация, но не с команды, на которую указывает точка возврата предыдущей активации в динамической цепи вызовов, а с команды, на которую указывает оператор перехода goto. В зависимости от деталей определения и реализации языка этот процесс может оказаться достаточно сложным для его корректной реализации, особенно если принять во внимание конечные значения параметров, передаваемых по результату и по значению-результату в такой цепи прерванных вызовов подпрограмм.

9.4. Явно определяемая общая среда

Явная организация общей среды для совместного использования объектов данных — наиболее простой способ реализации совместного использования данных.

Множество объектов данных, которые будут совместно использоваться некоторым множеством подпрограмм, располагается в отдельном именованном блоке памяти. В каждой подпрограмме создается объявление, явным образом именующее этот блок. Тогда объекты данных, размещенные в этом блоке, становятся видимы из подпрограммы, и на них можно ссылаться обычным способом по имени. Этот блок в разных языках называется по-разному: в FORTRAN — это *блок* COMMON, в Ada — определенная форма *пакета*; в языке C отдельные переменные, отмеченные с помощью ключевого слова extern, совместно используются несколькими подпрограммами таким же способом. В C++ и в Smalltalk для этой цели можно использовать *классы*, хотя вообще-то они предназначены для других целей (см. главу 7). Наиболее подходящим термином для предмета нашего обсуждения является *общая среда*.

Спецификация. Общая среда идентична локальной среде для подпрограммы за тем исключением, что она не является частью какой-либо одной подпрограммы. В ней могут содержаться определения переменных, констант и типов, но не допускаются определения подпрограмм или формальных параметров. Спецификация *пакета* в языке Ada является примером такой среды:

```
package Shared_Tables is
   Tab_Size: constant integer := 100:
   type Table is array (1 .. Tab_Size) of real:
   Table1. Table2: Table:
   Curr_Entry:integer range 1 .. Tab_Size:
end
```

В этой спецификации пакета определены тип (Table), константа (Tab_Size), две таблицы (объекты данных типа Table) и целая переменная (Curr_Entry), представляющие вместе группу объектов данных (и определений типов), которые используются несколькими подпрограммами. Определение пакета дается за пределами всех подпрограмм, которые используют эти переменные.

Если в подпрограмме P требуется получить доступ к общей среде, определенной этим пакетом, то в нее следует включить явный оператор with среди других объявлений:

```
with Shared_Tables:
```

Теперь любое имя, определенное в пакете Shared_Tables, можно использовать в теле подпрограммы P, как если бы оно было частью локальной среды этой подпрограммы. Для ссылки на эти имена мы используем уточненное имя в виде имя_пакета.имя_переменной. Таким образом, в подпрограмме P мы можем написать:

```
Shared_Tables.Table1(Shared_Tables.Curr_Entry) :=
      Shared_Tables.Table2(Shared_Tables.Curr_Entry) + 1:
```

без задания каких-либо дополнительных объявлений для любого из этих имен. Имя пакета должно предшествовать имени каждой переменной, так как одна и та же подпрограмма может использовать много пакетов, в некоторых из них могут быть объявлены одинаковые имена. (Если таких конфликтов не происходит, то в языке Ada имя пакета можно опустить, если использовать оператор use — например, with Shared_Tables; use Shared_Tables.) Вообще говоря, любое количество подпрограмм может использовать одну и ту же общую среду включением оператора

with Shared_Tables, и одна подпрограмма может использовать любое количество общих сред.

Реализация. В C и FORTRAN каждая подпрограмма, использующая общую среду, должна также включать в себя объявления для всех совместно используемых переменных, чтобы компилятор знал соответствующие объявления, хотя сама общая среда также объявлена где-то в другом месте подпрограммы. В языке Ada предполагается, что компилятор получает объявление общей среды из библиотеки или другой части текста программы, когда во время компиляции подпрограммы он встречает оператор with.

Объявления для общей среды добавляются к таблице символов компилятора как дополнительное множество локальных имен, на которые можно ссылаться в подпрограмме. Затем для целей статической проверки типов и генерации выполняемого кода каждая ссылка на имя в теле подпрограммы отыскивается в этой таблице обычным способом.

Во время выполнения подпрограммы общая среда представлена блоком памяти, содержащим объекты данных, объявленные в ее определении. Поскольку эти объекты потенциально могут принадлежать к различным типам данных, а их объявления известны во время компиляции, то этот блок можно рассматривать как *запись*. Имя блока представляет собой имя записи, а отдельные переменные в блоке — компоненты записи. Ссылки на отдельные переменные в блоке затем преобразуются в обычные адреса компонентов записи, вычисляемые по формуле «базовый адрес + смещение».

Блок памяти, представляющий общую среду, должен существовать в памяти, пока потенциально может быть вызвана любая подпрограмма, использующая этот блок, поскольку каждая активация любой такой подпрограммы должна иметь к нему доступ. Поэтому обычно блок располагается в памяти статически, так что он создается в начале выполнения программы и остается в памяти до ее завершения. В памяти может быть размещено множество таких общих блоков вперемешку с блоками, представляющими сегменты кодов подпрограмм, и другими статически размещенными блоками для переменных.

Подпрограмма, ссылающаяся на объект данных из общего блока, должна знать базовый адрес этого блока. Простым способом реализации этого требования является размещение указателя на блок (то есть его базового адреса) в сегменте кода. Эти связи через указатели между подпрограммой и общими блоками, которые она использует, аналогичны связям между подпрограммой и сегментами кода вызываемых ею подпрограмм. Одной из главных задач редактора связей, который перед началом выполнения программы собирает в памяти подпрограммы и общие блоки, является запоминание действительных указателей связи на эти блоки в каждом сегменте кода, который потребуется в процессе выполнения. Эта структура показана на рис. 9.9. Во время выполнения подпрограммы ссылка на объект данных из общего блока обрабатывается следующим образом: для вычисления действительного местоположения этого объекта данных в памяти из сегмента кода подпрограммы берется базовый адрес соответствующего общего блока и к нему добавляется заранее вычисленное смещение для этого объекта данных.

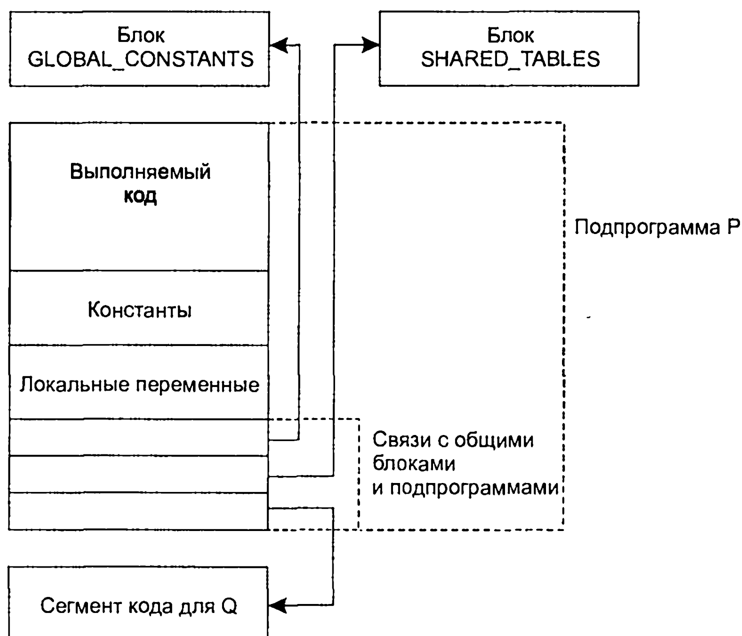


Рис. 9.9. Связь с общими блоками в подпрограммах

Совместное использование определяемых явно переменных

Другая форма явного определения совместно используемых объектов данных позволяет сделать объект данных в локальной среде какой-либо подпрограммы видимым для других подпрограмм. Так, вместо создания группы переменных в общей среде, отделенной от любой подпрограммы, мы используем тот факт, что каждая переменная имеет своего «владельца» — подпрограмму, в которой она объявляется. Чтобы сделать эту локальную переменную видимой вне определения «владеющей» ею подпрограммы, следует задать явное *определение ее экспорта*, подобное объявлению defines в следующем коде:

```
procedure P(...);
   defines X, Y, Z;          X, Y и Z становятся доступными для экспорта
   X, Y, Z: real;            Обычные объявления для X, Y и Z
   U, V: integer;            Другие локальные переменные
begin ...end;                Операторы
```

Если какой-либо другой подпрограмме требуется получить доступ к экспортированной переменной, то в ней используется явное *определение импорта* для импортирования переменной (например, включением объявления uses, где указаны и имя экспортированной переменной, и имя подпрограммы, откуда она экспортирована):

```
procedure Q(...);
   uses P.X, P.Z;            Импорт X и Z из подпрограммы P
   Y: integer;               Другие объявления
begin...end                  В операторах можно ссылаться на X и Z
```

Описанная модель используется в языке C, где в объявлении переменных может задаваться модификатор extern.

Реализация. Эффект от такого способа совместного использования переменных мало чем отличается от использования переменных в общей среде. Экспортированная локальная переменная должна оставаться в памяти в промежутках между активациями подпрограммы, в которой она была определена. Поэтому обычно она хранится в сегменте кода этой подпрограммы, как обычная локальная сохраняемая переменная. При ссылке на такую переменную из другой подпрограммы, которая ее импортирует, используется базовый адрес сегмента кода экспортирующей подпрограммы и к нему добавляется соответствующее смещение.

9.4.1. Динамическая область видимости

Альтернативой использованию явно определяемых общих сред для совместно используемых данных является ассоциация *нелокальной среды* с каждой выполняющейся подпрограммой. Нелокальная среда для подпрограммы P состоит из множества локальных сред активаций других подпрограмм, которые становятся доступными для P во время ее выполнения. Если в P имеется ссылка на переменную X, для которой нет локальной ассоциации, тогда для определения ассоциации для X используется среда нелокальных ссылок. Что представляет собой нелокальная среда для P? В блочно-структурированном языке правила статической области видимости определяют неявную нелокальную среду для каждой подпрограммы. Этот довольно сложный способ рассматривается в следующем разделе. Более простой, но менее распространенной альтернативой является использование сред локальных ссылок для подпрограмм в текущей динамической цепи, которую мы и рассмотрим в данном разделе.

Предположим, что в некотором языке программирования среда локальных ссылок разрушается по окончании работы подпрограммы, а определения подпрограмм не могут быть вложенными друг в друга. Каждая подпрограмма, таким образом, определена отдельно от других. В этом случае, характерном для языков LISP, APL и SNOBOL4, не существует статической структуры программы, на которой основываются правила области видимости для разрешения ссылок на *нелокальные* идентификаторы. Если, например, подпрограмма P содержит ссылку на имя X и X локально не определено в P, то какое из определений X в других подпрограммах следует использовать? Естественный ответ заключается в том, что нужно рассмотреть *динамическую цепь* активаций подпрограмм, которая приводит к активации P. Рассмотрим процесс выполнения программы: предположим, что главная программа вызывает подпрограмму A, которая вызывает B, которая, в свою очередь, вызывает P. Если в P имеется ссылка на X, но отсутствует ассоциация для этой переменной, то естественно было бы обратиться к подпрограмме B, вызвавшей P, и посмотреть, нет ли в ней ассоциации для X. Если такая ассоциация имеется, то она и используется в подпрограмме P. В противном случае мы обращаемся к подпрограмме A, вызвавшей B, и проверяем, имеется ли в A ассоциация для X. Мы использовали *последнюю из созданных ассоциаций* для X в динамической цепи вызовов подпрограмм, приводящей к P. Такое определение значения нелокальной ссылки называется *правилом*

последней ассоциации; это — правило определения ссылки, основанное на динамической области видимости.

Если среда нелокальных ссылок определена с помощью правила последней ассоциации, никаких правил статической области видимости не используется (то есть во время трансляции программы не предпринимается никаких действий для поиска определения, ассоциированного со ссылкой на какой-либо идентификатор, для которого отсутствует локальное определение). Во время выполнения программы, когда объект данных с именем X создается как часть активации подпрограммы P, *динамической областью* видимости ассоциации для X становятся все активации подпрограмм, вызванных из подпрограммы P или из этих вызванных подпрограмм и т. д. Имя X видимо в этой динамической области видимости (за исключением активаций, где оно скрыто более поздней подпрограммой, которая имеет свою собственную ассоциацию для X). Если взглянуть на эту ситуацию с другой стороны, то *нелокальная среда* активации подпрограммы P состоит из всей динамической цепи активаций подпрограмм, приводящей к подпрограмме P.

Причиняющим наибольшее беспокойство свойством такой нелокальной среды является то, что она может *измениться* между активациями подпрограммы P. Так, для одной активации P, когда создается нелокальная ссылка на X, может оказаться, что самая последняя в цепи вызовов подпрограмм ассоциация для X представляет его как имя массива. Для второй активации P, реализованной через другую последовательность предшествующих вызовов подпрограмм, динамическая цепь может измениться, так что в самой последней ассоциации для X этому имени будет соответствовать строка символов. Для третьей активации P может вообще не оказаться ассоциации для X в вызывающей цепи, так что ссылка на X окажется ошибкой. Эта изменчивость ассоциации для X означает, что требуется применять *динамическую проверку типов*. Поэтому этот метод используется только в языках, подобных LISP, APL и SNOBOL4, где динамическая проверка типов используется по другим причинам.

Реализация. Реализация правила последней ассоциации для нелокальных ссылок не вызывает затруднений, если для хранения записей активаций использовать центральный стек. Среда локальных ссылок для каждой подпрограммы представляется как часть ее записи активации. При входе в подпрограмму создается запись активации, при выходе из нее запись активации уничтожается.

Предположим, что подпрограмма P вызывает подпрограмму Q, которая, в свою очередь, вызывает R. Когда выполняется подпрограмма R, центральный стек может выглядеть примерно так, как изображено на рис. 9.10. Для разрешения нелокальной ссылки на X просматривается стек, начиная с локальной среды для подпрограммы R, в обратном направлении через все ассоциации, находящиеся в стеке, пока не будет обнаружена самая последняя созданная ассоциация для X. Как показано на рисунке, некоторые ассоциации в стеке будут скрыты более поздними ассоциациями для того же самого идентификатора.

Однако цена такой реализации правила последней ассоциации довольно высока. Поиск, необходимый для разрешения каждой нелокальной ссылки, требует времени и снова приводит к необходимости хранения некоторого представления *идентификаторов* в таблицах локальных ассоциаций, так как в каждой локальной таблице позиция ассоциации для X может быть различной. Таким образом, нельзя применить формулу «базовый адрес + смещение» для разрешения нелокальной ссылки.

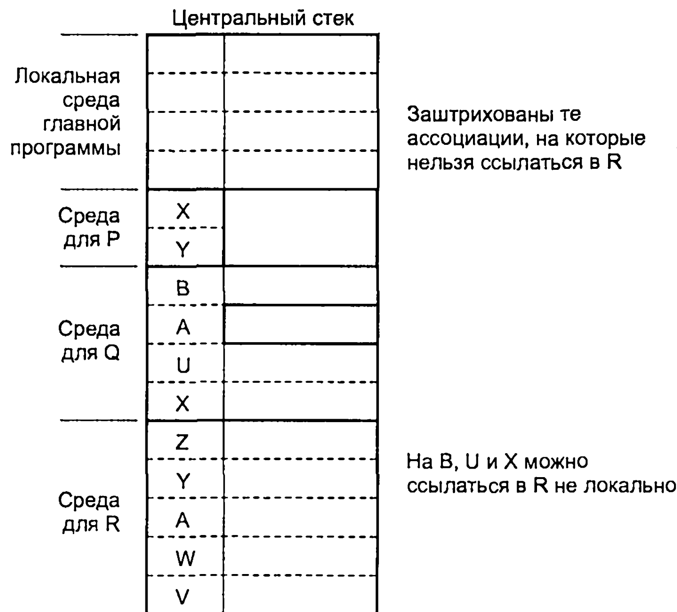


Рис. 9.10. Активная среда ссылок во время выполнения программы

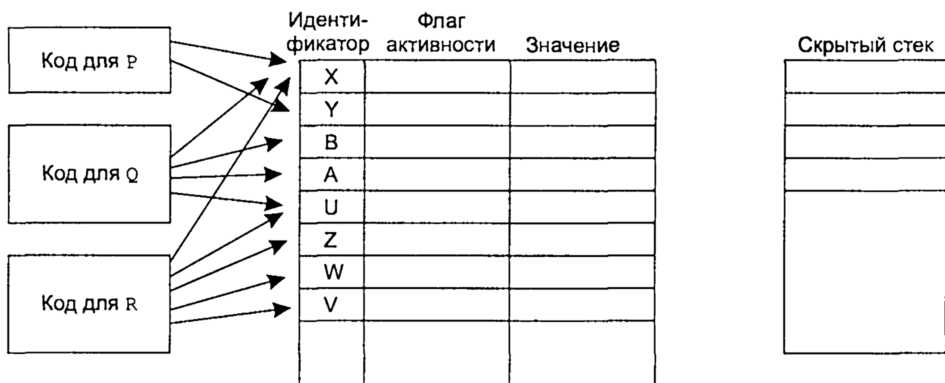
Как можно избежать процедуры поиска в разрешении нелокальных ссылок? Возможен компромисс между ценой разрешения не локальных ссылок и ценой входа и выхода из подпрограммы, который может оказаться выгодным, если предположить, что разрешение нелокальных ссылок происходит значительно чаще, чем вход и выход из подпрограмм (то есть если нелокальная среда будет использоваться чаще, чем она модифицируется).

Альтернативная реализация использует центральную таблицу, общую для всех подпрограмм, — *центральную таблицу среды ссылок*. Структура центральной таблицы такова, что во время выполнения программы в ней содержатся *все активные на данный момент ассоциации идентификаторов* независимо от того, являются они локальными или нелокальными. Если мы для простоты предположим, что множество идентификаторов, ссылки на которые имеются хотя бы в одной подпрограмме, может быть определено во время трансляции, тогда при инициализации центральной таблицы в ней содержится по одной записи для каждого идентификатора, независимо от количества различных подпрограмм, в которых этот идентификатор появляется. Каждая запись в таблице также содержит *флажок активности*, который указывает, имеется ли в данный момент для конкретного идентификатора активная ассоциация и, кроме того, отведено ли в записи место для указателя на объект ассоциации.

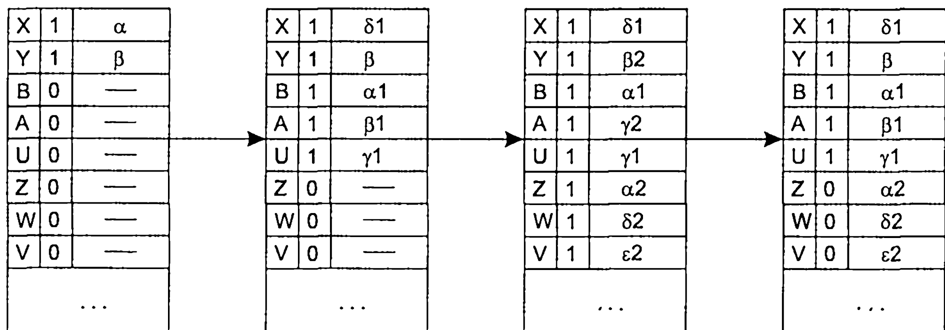
Процедура разрешения всех ссылок в подпрограммах обращается к этой центральной таблице, используя схему «базовый адрес + смещение», описанную выше. Поскольку текущая ассоциация для идентификатора X всегда расположена в одном и том же месте в центральной таблице независимо от подпрограммы, в которой появилась ссылка на этот идентификатор, и независимо от того, является ли ссылка локальной или нелокальной, то можно применять такое простое вычисле-

ние для разрешения ссылки. Для каждой ссылки требуется всего лишь проверить флажок активации соответствующей записи таблицы, чтобы убедиться, что эта ассоциация в данный момент активна. Использованием центральной таблицы мы, таким образом, достигли желаемой цели разработать относительно эффективный процесс разрешения нелокальных ссылок без процедуры поиска.

ДОСТУП ЧЕРЕЗ ЦЕНТРАЛЬНУЮ ТАБЛИЦУ И СКРЫТЫЙ СТЕК



ЦЕНТРАЛЬНАЯ ТАБЛИЦА



X	1	α
Y	1	β
B	0	—
A	0	—
U	0	—
Z	0	—
W	0	—
V	0	—
...		

X	1	δ1
Y	1	β
B	1	α1
A	1	β1
U	1	γ1
Z	0	—
W	0	—
V	0	—
...		

X	1	δ1
Y	1	β2
B	1	α1
A	1	γ2
U	1	γ1
Z	1	α2
W	1	δ2
V	1	ε2
...		

X	1	δ1
Y	1	β
B	1	α1
A	1	β1
U	1	γ1
Z	0	α2
W	0	δ2
V	0	ε2
...		

СКРЫТЫЙ СТЕК

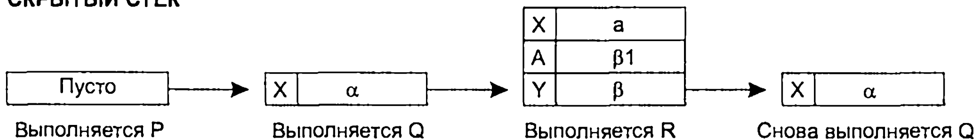


Рис. 9.11. Центральная таблица среды ссылок для разрешения нелокальных ссылок

Вход и выход из подпрограмм — более дорогостоящие операции, поскольку каждое изменение в среде ссылок требует модификации центральной таблицы. Когда подпрограмма P вызывает подпрограмму Q, центральная таблица должна быть изменена, чтобы отразить новую локальную среду для подпрограммы Q. Таким образом, каждая запись таблицы, соответствующая какому-то локальному идентификатору для Q, должна быть модифицирована, чтобы отразить новую локальную ассоциацию для Q. В то же время, если старая запись таблицы для идентификатора

была активной, она должна быть сохранена, чтобы ее можно было снова активизировать, когда завершится выполнение Q и управление перейдет снова к подпрограмме P. Поскольку записи, требующие модификации, с большой вероятностью окажутся рассеянными по центральной таблице, то эту модификацию нужно осуществлять постепенно, запись за записью. При выходе из Q те ассоциации, которые были деактивированы и сохранены при входе в Q, должны быть восстановлены и снова активированы. При этом снова, как и в описанных ранее моделях, требуется стек времени выполнения, но здесь он используется как *скрытый стек* для хранения деактивированных ассоциаций. Так как при входе в Q каждая ассоциация для локальных идентификаторов изменяется, то старая ассоциация помещается как блок в скрытый стек. После возврата из Q ассоциации из верхнего блока стека восстанавливаются в соответствующих местах центральной таблицы. Моделирование с помощью центральной таблицы представлено на рис. 9.11. Дополнительное преимущество использования центральной таблицы возникает, если в языке запрещается генерировать новые ссылки во время выполнения. В этом случае, как и в рассмотренном ранее случае с локальными таблицами, идентификаторы могут быть удалены из таблицы, поскольку они никогда не будут использоваться, их заменяют вычисления по схеме «базовый адрес + смещение». (Имеется в виду, что идентификатор во время выполнения представлен просто своим смещением в таблице.)

9.4.2. Статическая область видимости и блочная структура

В таких языках, как Pascal и Ada, в которых программы имеют блочную структуру, обработка нелокальных ссылок на совместно используемые данные оказывается более сложной. Если вы снова прочитаете о правилах статической области видимости, обсуждавшиеся в разделе 9.2.3, то вы заметите, что каждая ссылка на идентификатор внутри подпрограммы связана с определением этого идентификатора в тексте программы, даже если он не является локальным для нее. Таким образом, среда нелокальных ссылок для каждой подпрограммы во время ее выполнения уже определена правилами статической области видимости во время трансляции программы. Задача реализации заключается в обеспечении согласованности между правилами статической и динамической областей видимости, чтобы нелокальная ссылка во время выполнения программы была корректно связана с объектом данных, соответствующим определению этого идентификатора в тексте программы.

Листинг 9.10 иллюстрирует применение правил статической области видимости к блочно-структурированной программе на языке Pascal. Подпрограмма R вызывается из подпрограммы Q, которая, в свою очередь, вызывается подпрограммой P. Переменная X определена в P, Q и в главной программе. Ссылка на X в подпрограмме R является, таким образом, нелокальной. Правила статической области видимости определяют эту ссылку как ссылку на переменную X, которая определена в главной программе, а не в подпрограммах P и Q. Значение нелокальной ссылки на X не зависит от конкретной динамической цепи вызовов подпрограмм, которая приводит к активации R, в отличие от правила последней ассоциации из предыдущего раздела, которое соотносит X с определением X, данным в Q или P — в зависимости от того, откуда вызвана подпрограмма R.

Правила статической области видимости легко реализуются в компиляторе. При обработке каждого определения подпрограммы компилятором создается локальная таблица объявлений и добавляется в цепь таких же локальных таблиц, которые представляют локальные среды главной программы и других подпрограмм, в которые вложена обрабатываемая подпрограмма. Таким образом, при компиляции подпрограммы R компилятор добавляет локальную таблицу объявлений для R к цепи, содержащей только определение главной программы. Во время компиляции эта цепь просматривается в поисках объявления переменной X, начиная от локальных объявлений для подпрограммы R через локальные объявления для подпрограмм, в которые она вложена, и заканчивая объявлениями для главной программы. По завершении компиляции подпрограммы R компилятор удаляет из этой цепи локальную таблицу для R. Заметим, что здесь имеется определенное сходство с поиском значения для X при использовании правила последней ассоциации, описанного в разделе 9.4.1. Но в данном случае поиск объявления для X осуществляется не во время выполнения, а во время *компиляции*. Цепи локальных таблиц объявлений отражают статическую вложенность определений подпрограмм в тексте программы, а не динамическую цепь вызовов подпрограмм во время выполнения программы.

Листинг 9.10. Процедура на Pascal с нелокальными ссылками

```
program Main;
    var X, Y: integer;
    procedure R;
       var Y: real;
       begin
       ...
       X := X+1;       { Нелокальная ссылка на X }
       ...
       end {R};
    procedure Q;
       var X: real;
       begin
       ...
       R;              { Вызов процедуры R }
       ...
       end {Q};
    procedure P;
       var X: Boolean;
       begin
       ...
       Q;              { Вызов процедуры Q }
       ...
       end {P};
    begin              { начало Main }
    ...
    P;                 { Вызов процедуры P }
    ...
    end.
```

В блочно-структурированном языке во время выполнения программы центральный стек используется для хранения записей активации подпрограмм. Локальная среда для каждой подпрограммы хранится в ее записи активации. Сложность поддержания статической области видимости при использовании правил динамической области видимости становится очевидной из рис. 9.12, где показано содержи-

мое центрального стека во время выполнения подпрограммы R из листинга 9.10. Когда при выполнении R встречается нелокальная ссылка на X, то операция обработки ссылки должна найти ассоциацию для X в главной программе, а не в подпрограмме Q, откуда была вызвана R. Но, к сожалению, простой поиск вниз по стеку приводит к ассоциации для X, содержащейся в подпрограмме Q. Проблема заключается в том, что последовательность локальных таблиц в стеке отражает *динамическую вложенность активаций* подпрограмм — вложенность, основанную на цепи последовательных вызовов подпрограмм во время выполнения программы. Но теперь именно *статическая вложенность определений* подпрограмм определяет нелокальную среду, а текущая структура стека не содержит никакой информации об этой статической вложенности.

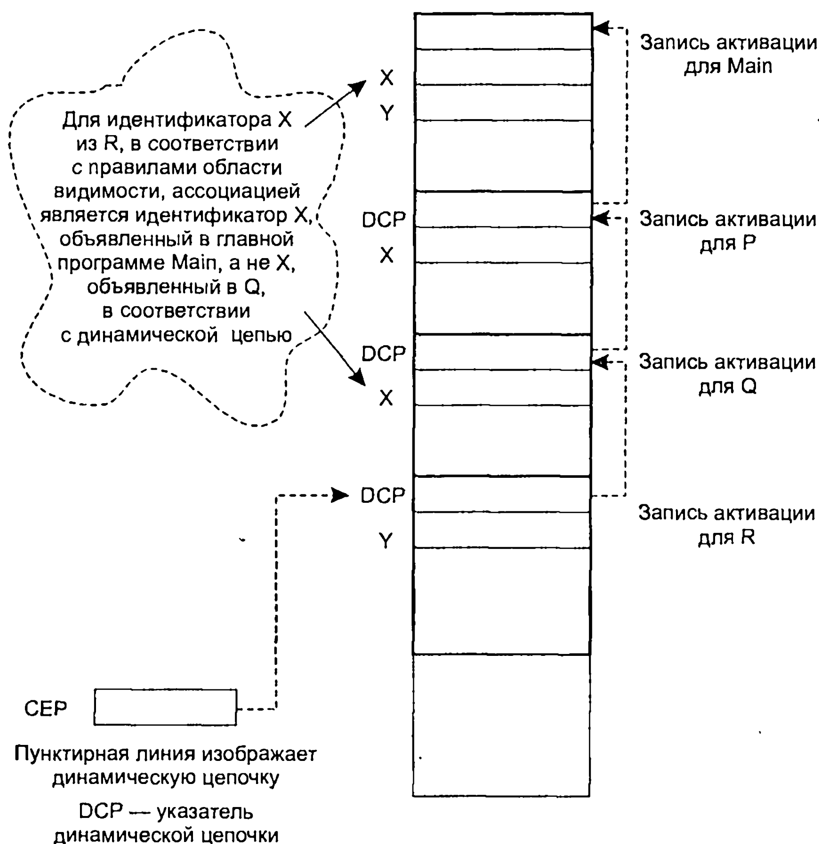


Рис. 9.12. Неполный центральный стек во время выполнения, использующего статическую область видимости

Для завершения реализации необходимо представить статическую блочную структуру во время выполнения таким образом, чтобы ее можно было использовать для управления процессом разрешения нелокальных ссылок. Заметим, что во многих отношениях правило разрешения нелокальных ссылок в этом случае аналогично разрешению нелокальных ссылок с использованием правила последней

ассоциации: чтобы найти ассоциацию, соответствующую ссылке на X, осуществляется поиск в цепи таблиц локальных сред, пока не будет найдена ассоциация для X. Но эта цепь составлена не из *всех* локальных таблиц, в данный момент находящихся в стеке, а только из тех, которые *представляют блоки или подпрограммы, определения которых статически включают в себя определение текущей подпрограммы в исходном тексте программы.* Тогда поиск по-прежнему осуществляется вниз по стеку среди некоторых таблиц, содержащихся в нем, но только тех, которые действительно являются частью среды ссылок.

Реализация статической цепи. Приведенные выше наблюдения подводят нас к наиболее прямой реализации корректной среды ссылок — методу *статической цепи*. Предположим, что мы несколько модифицировали таблицы локальных сред в стеке таким образом, что теперь каждая таблица начинается со специального элемента — *указателя статической цепи*. Этот указатель всегда содержит базовый адрес другой локальной таблицы, расположенной ниже в стеке. Таблица, на которую указывает этот указатель, — это таблица, представляющая локальную среду того блока или подпрограммы, которая статически включает в себя рассматриваемую подпрограмму. (Разумеется, поскольку каждая таблица локальной среды является просто частью записи активации, мы можем использовать базовый адрес записи активации вместо базового адреса локальной среды.)

Указатели статических цепей являются основой для простой схемы разрешения ссылок. Для разрешения ссылки на X мы следуем по указателю CEP и попадаем в текущую локальную среду, расположенную в вершине стека. Если в этой локальной среде не найдено ассоциации для X, то по указателю статической цепи мы переходим во вторую среду ссылок, расположенную ниже в стеке. Если ассоциация для X и там не найдена, то мы продолжаем поиск дальше вниз по стеку, переходя с помощью указателей статической цепи от одной среды ссылок к другой, пока не будет обнаружена локальная таблица, содержащая ассоциацию для X. Первая найденная таким образом ассоциация и будет правильной ассоциацией для X. Рис. 9.13 иллюстрирует статическую цепь для программы из листинга 9.10.

Хотя кажется, что при этом требуется осуществлять *поиск* в каждой статически связанной среде до тех пор, пока не будет найдена ассоциация для X, на самом деле это не обязательно так. Поскольку компилятор создает таблицу локальной среды для каждого локального объявления (см., например, табл. 9.1), он отслеживает, какие подпрограммы статически вложены в данную среду. Для каждой ссылки на X компилятор *подсчитывает* количество тех включающих сред, в которых (во время компиляции) следует искать правильную ассоциацию для X. Затем он генерирует код перехода по указателю статической цепи столько раз, сколько насчитано включающих сред, чтобы достичь той записи активации, в которой содержится ассоциация для переменной X. Это позволяет избежать хранения имени идентификатора в стеке во время выполнения, как отмечалось ранее.

Описанная схема является исключительно эффективной реализацией, обычно требующей n обращений к указателю статической цепи для доступа к переменной, которая объявлена на уровне k, а используется на уровне $n + k$. В большинстве языков программирования глубина вложенности подпрограмм не превышает 4–5 уровней (типичная имеет 2–3 уровня), поэтому значение n редко превышает 1 или 2, но в таких языках, как, например, Pascal, глубина уровней вложенности может быть

произвольной. Однако мы можем сделать еще лучше — организовать доступ к правильной среде, выполнив всего одну ссылку на указатель статической цепи, как кратко описано ниже.

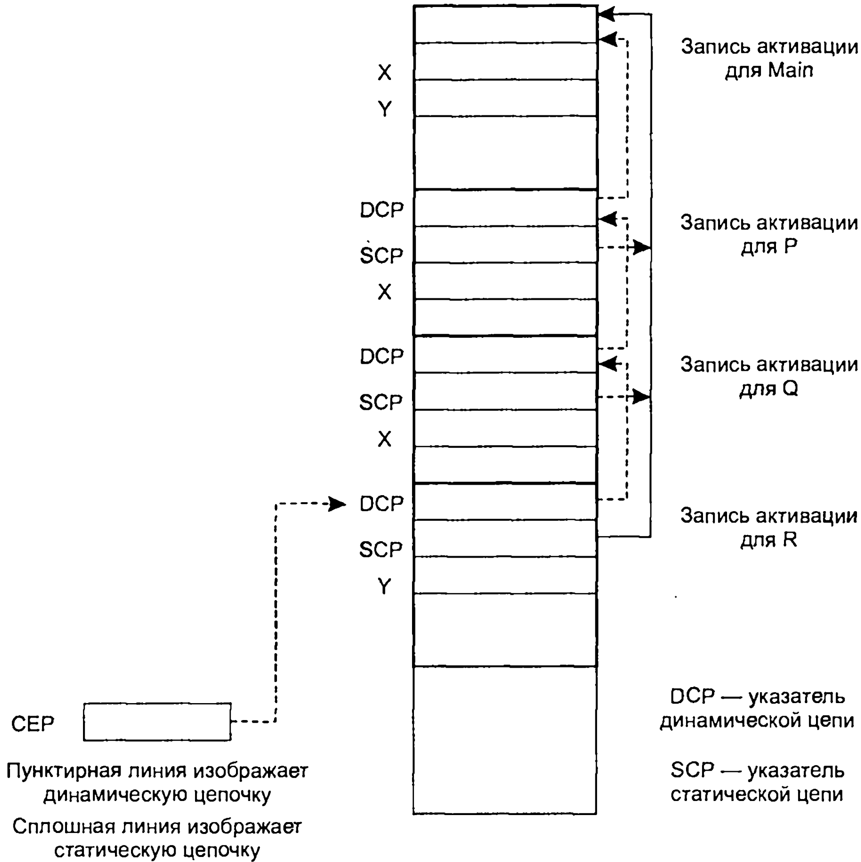


Рис. 9.13. Центральный стек во время выполнения программы

Метод статической цепи значительно упрощает вход и выход из подпрограмм. Когда вызывается какая-либо подпрограмма, на вершине стека создается ее запись активации. В этот момент в ней должен быть размещен соответствующий указатель статической цепи, указывающий на запись активации, расположенную ниже в стеке. При выходе из подпрограммы требуется только удалить из стека запись активации обычным образом; не нужно выполнять никаких специальных действий, связанных с указателем статической цепи.

Как мы можем определить соответствующий указатель статической цепи, который следует установить при входе в подпрограмму? Предположим, что подпрограмма R вызвана из Q и R определена в тексте исходной программы, непосредственно вложенной в Main, как на рис. 9.12. Когда во время выполнения программы происходит вход в R, то соответствующий указатель статической цепи указывает назад на запись активации подпрограммы Main. В точке вызова на вершине стека нахо-

дится запись активации подпрограммы Q, и запись активации подпрограммы R должна быть помещена на вершину стека над записью активации подпрограммы Q. Как же определить, что правильный указатель статический цепи — это указатель на Main? Обратите внимание на то, что ссылка на идентификатор R, имя подпрограммы, в операторе вызова подпрограммы R в Q является нелокальной. Если определено (во время компиляции), что идентификатор R должен быть объявлен на один уровень вложенности раньше расположения оператора вызова подпрограммы R в Q, тогда во время выполнения указатель статической цепи для R должен указывать на запись активации, расположенную на один шаг впереди[1] относительно расположения записи активации подпрограммы Q в статической цепи. Таким образом, в точке вызова подпрограммы R в Q указатель статической цепи для R можно определить как указатель на запись активации подпрограммы Main, так как именно ее запись активации расположена на один шаг впереди относительно записи активации подпрограммы Q в статической цепи. Эта ситуация в точности соответствует той, при которой имя R было бы определено как локальная переменная в Main, на которую существовала бы нелокальная ссылка в подпрограмме Q, за исключением того, что после перехода по статической цепи от Q до соответствующей записи активации не требуется добавлять смещение. Вместо этого найденная запись активации становится целевым объектом указателя статической цепи подпрограммы R после создания ее записи активации.

Реализация дисплея. Чтобы разобраться, как можно улучшить реализацию статической цепи для доступа к нелокальным ссылкам, нам понадобится несколько предварительных наблюдений.

1. Для любой подпрограммы R во время ее выполнения (когда ее таблица локальной среды находится на вершине стека) длина статической цепи, ведущей от локальной таблицы подпрограммы R вниз по стеку (и в конечном счете к таблице главной программы), *постоянна* и просто равна глубине статической вложенности определения подпрограммы R в исходной программе во время компиляции. Эта длина не меняется во время выполнения программы. Если, например, определение R содержится в блоке, который расположен непосредственно внутри самого внешнего блока программы, то длина статической цепи для R во время выполнения всегда равна 3 — в нее входят локальная таблица для R, таблица для содержащего R блока и таблица для самого внешнего блока (главная программа). На рис. 9.13 и в листинге 9.10, например, длина статической цепи для R всегда равна 2.
2. В этой цепи постоянной длины нелокальная ссылка всегда будет разрешаться точно в одной и той же точке цепи. Например, на рис. 9.13 нелокальная ссылка на имя X из подпрограммы R всегда будет разрешена во второй таблице цепи. Этот факт является простым следствием статической структуры программы. Количество уровней статической вложенности, которые необходимо пройти от определения подпрограммы R, чтобы найти объявление для X, фиксируется во время компиляции и затем не изменяется.

---

[1] Напомним, что в статической цепи первой является запись активации текущей выполняемой программы, а последней — запись активации главной программы. Поэтому по статической цепи мы движемся вперед! — *Примеч. науч. ред.*

3. Позиция в цепи, где будет разрешена нелокальная ссылка, может быть определена при компиляции. Этот факт мы использовали при подсчете количества указателей статической цепи, которое нам надо было пройти, в рассмотренной ранее реализации статической цепи. Например, во время компиляции мы можем определить, что во время выполнения ссылка на X в R будет найдена во второй таблице вперед по статической цепи. Кроме того, во время компиляции нам известно относительное местоположение X в этой локальной таблице. Так, например, во время компиляции мы можем сделать вывод, что во время выполнения ассоциация для X будет второй записью во второй таблице, отсчитываемой вперед по статической цепи.

Базис для разработки эффективной операции обработки ссылок теперь очевиден. Вместо того чтобы явным образом осуществлять поиск идентификатора вперед по статической цепи, нужно только пропустить в ней фиксированное количество таблиц, а затем применить формулу «базовый адрес + смещение» для выбора нужной записи таблицы. Во время выполнения мы представляем идентификатор парой значений (позиция в цепи, смещение). Например, если идентификатор X, ссылка на который встречается в подпрограмме R, находится в третьей впереди по цепи записи первой таблицы, тогда в скомпилированном коде для R идентификатор X может быть представлен парой (1, 3). Такое представление позволяет использовать довольно простой алгоритм разрешения ссылок.

В этой реализации текущая статическая цепь копируется в отдельный вектор, называемый дисплеем, на входе в каждую подпрограмму. Дисплей хранится отдельно от центрального стека и часто реализуется с помощью быстрых регистров. В каждой конкретной точке во время выполнения дисплей содержит такую же последовательность указателей, которая встречается и в статической цепи подпрограммы, выполняемой в данный момент. Рис. 9.14 иллюстрирует дисплей для программы, приведенной в листинге 9.10.

Разрешение ссылок с использованием дисплея чрезвычайно просто. Применим несколько модифицированное представление идентификаторов во время выполнения. Мы по-прежнему будем использовать пары целых чисел, но теперь первое число в паре (например, 3 в (3, 2)) будет означать количество шагов, которые надо сделать назад от *конца* цепи до соответствующей записи активации (а не вперед от начала цепи, как было раньше). Второе число в этой паре по-прежнему представляет собой смещение в записи активации. Теперь, если нелокальная ссылка представлена в виде (3, 10), то соответствующая ассоциация находится за два шага.

1. Первый элемент пары (то есть число 3) будем считать индексом в дисплее. Оператор `base_adress = display[3]` присваивает переменной `base_adress` указатель на базовый адрес соответствующей записи активации.

2. Позицию искомой записи таблицы вычисляем как «базовый адрес + смещение», в данном случае `base_adress + 10`.

Обычно эти два шага объединяют в один, используя косвенную адресацию через элементы дисплея. Если во время выполнения дисплей реализован в быстрых регистрах, то при разрешении каждой ссылки на идентификатор требуется только одно обращение к памяти.

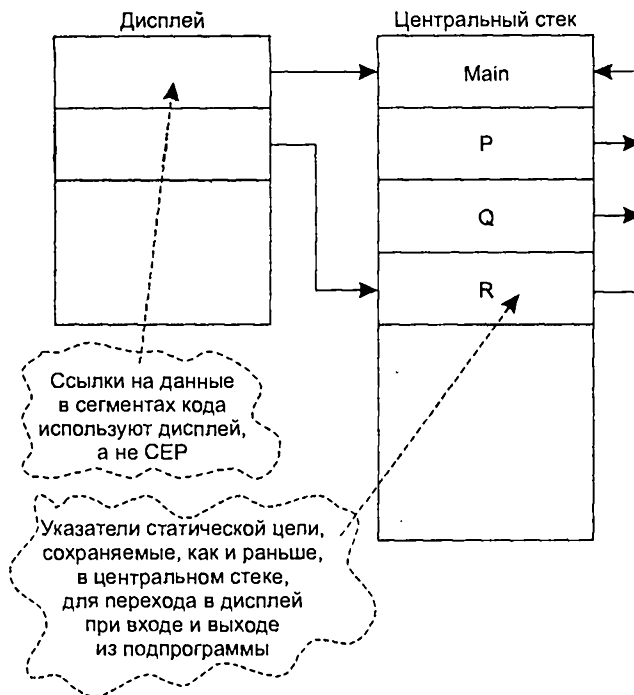


Рис. 9.14. Центральный стек и дисплей во время выполнения

Хотя использование дисплея упрощает операцию разрешения ссылки, оно усложняет вход и выход из подпрограмм, так как при каждом входе или выходе дисплей должен изменяться таким образом, чтобы в любой момент отражать *текущую активную* статическую цепь. Простейшей процедурой является поддержка указателей статической цепи в центральном стеке, как описывалось ранее, и перезагрузка дисплея соответствующими указателями статической цепи при каждом входе и выходе из подпрограммы с использованием команд, вставленных компилятором в пролог и эпилог сегмента кода каждой подпрограммы.

Объявления в локальных блоках

В языках, подобных C, допускается объявлять локальные переменные для блока операторов, вложенного в процедуру (листинг 9.11). На первый взгляд кажется, что каждому блоку требуется своя запись активации для хранения этих переменных. Но существует важное различие между этими блоками и описанными выше записями активаций процедур. Динамическая природа вызовов процедур является причиной того, что вообще мы никогда не можем быть уверены, какие процедуры выполняются в данный момент времени.

Таким образом, если бы каждый блок из листинга 9.11 представлял собой некоторую процедуру, могло бы случиться, что блок, содержащий k, и блок, содержа-

щий m, выполнялись бы одновременно (например, блок, содержащий k, вызывает блок, содержащий m). Поэтому для таких блоков потребовались бы отдельные записи активации.

Листинг 9.11. Локальные объявления в С

```
real procl( параметры )
      {int i.j:
      ...              /* операторы */
        {int k, 1: .../* операторы */ }
        {int m. n:
        ...            /* операторы */
        {int x: ...    /* операторы */ }
        {int y: ...    /* операторы* / }
        } }
```

Но в рассматриваемом примере на С такая ситуация невозможна. Если мы находимся внутри блока, содержащего переменную k, мы не можем находиться в области видимости переменной m. Аналогично, если мы находимся в области видимости x, мы не можем находиться в области видимости y. Это позволяет применить простую стратегию распределения памяти, аналогичную той, которая применяется для структур хранения вариантных записей (см. рис. 6.9). Для хранения переменных k и 1 можно использовать те же области памяти, что и для переменных m и n, поскольку они никогда не будут активизированы одновременно и все эти области расположены в области памяти, выделяемой для содержащей блоки операторов процедуры. Локальные объявления нужны только для того, чтобы определить область видимости (множество операторов), где каждая переменная может использоваться. Память выделяется для активации подпрограммы в целом. Структура записи активации, куда входят и описанные блоки, проиллюстрирована на рис. 9.15.

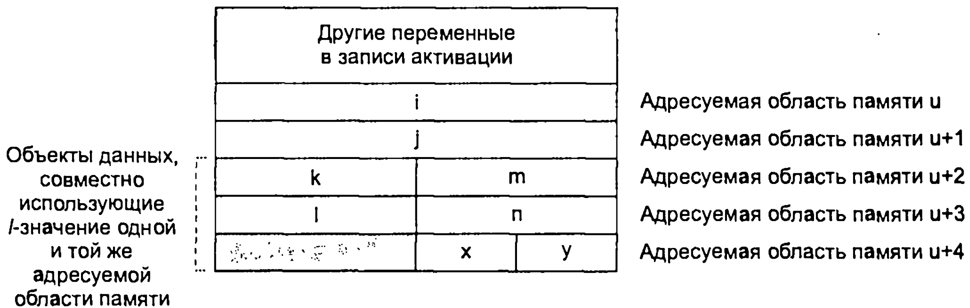


Рис. 9.15. Совмещение областей памяти для переменных в записи активации

9.5. Рекомендуемая литература

Большую часть упоминаемых в главе 1 книг и статей, которые посвящены структурам управления подпрограммами, можно порекомендовать и для изучения таких смежных проблем, как среды ссылок, совместно используемые данные и параметры. Тексты главы 3, посвященные написанию трансляторов, описывают, как компилировать такие структуры.

9.6. Задачи и упражнения

1. Когда среды локальных ссылок удаляются между активациями подпрограмм с помощью центрального стека, как, например, в языке С, иногда кажется, будто значения локальных переменных *сохраняются*. Например, в большинстве реализаций языка С, если в подпрограмме Sub имеется локальная переменная X и при первом вызове подпрограммы ей присваивается значение 5, тогда при втором вызове, если (случайно) сослаться на переменную X до того, как ей будет присвоено новое значение, то X может в этот момент по-прежнему иметь значение 5. Однако в той же самой программе при третьем вызове Sub можно обнаружить, что переменная X *не сохранила* значения, присвоенного ей во время второго вызова.
   а) Объясните эту кажущуюся аномалию: при каких обстоятельствах активация подпрограммы Sub, в которой есть ссылка на неинициализированную переменную X, может обнаружить, что X имеет прежнее значение, присвоенное ей во время предыдущего вызова подпрограммы Sub? При каких обстоятельствах значение переменной X, присвоенное ей в предыдущем вызове подпрограммы, не сохранится?
   б) Напишите простую программу на С и проследите за этими эффектами при ее выполнении в вашей локальной реализации языка.
2. Предположим, что некоторый язык позволяет при объявлении локальных переменных задавать их начальные значения, как, например, в следующем объявлении на языке Ada:
   ```
   X: integer := 50
   ```
   которое инициализирует переменную X, присваивая ей значение 50. Объясните два значения, которые может иметь эта инициализация в случаях:
   а) если локальные переменные *сохраняются* в промежутках между обращениями к подпрограмме;
   б) если локальные переменные *уничтожаются* между вызовами подпрограмм.
3. В Pascal (и многих других языках) допускается использовать *указатели* (переменные типа pointer) совместно с операцией new для создания новых объектов данных. Среда локальных ссылок в Pascal уничтожается в промежутках между вызовами подпрограмм. Комбинация этих двух свойств с большой вероятностью приводит к генерации *мусора* (недоступная информация, занимающая место в памяти) во время выполнения программы. Объясните, в чем состоит причина этого явления.
4. Возьмите какую-либо недавно написанную вами программу на языке, который использует *правила статической области видимости*, и для каждой содержащейся в ней подпрограммы перечислите входящие в ее среду имена:
   а) локальных ссылок;
   б) нелокальных ссылок;
   в) глобальных ссылок;
   г) предопределенных ссылок.

Затем решите обратную задачу: для имени, объявленного в каждой из этих сред ссылок, перечислите:

а) определения подпрограмм в его статической области видимости;

б) активации подпрограмм в его динамической области видимости.

5. При разработке подпрограмм часто бывает трудно обеспечить *сокрытие информации*, если в языке не допускается сохранение локальных данных между вызовами подпрограмм (как, например, в Pascal). Приведите пример ситуации в языке Pascal, когда удаление локальных сред приводит к необходимости сделать объект данных видимым за пределами подпрограммы, хотя доступ к этому объекту данных нужен только для этой подпрограммы.
6. Если в блочно-структурированном языке для разрешения нелокальных ссылок используется реализация с *дисплеем*, можно ли во время компиляции предсказать *максимальный размер* дисплея? Каким образом?
7. В языках, подобных Pascal, где используются правила статической области видимости и допускается рекурсивный вызов подпрограмм, иногда бывает трудно связать нелокальную ссылку в одной подпрограмме с конкретной переменной в другой подпрограмме, если у последней в данный момент времени существует несколько рекурсивных активаций. Например, пусть A, B, C и D — подпрограммы главной программы Main и A вызывается из Main. Предположим, что A вызывает B, B рекурсивно вызывает A, A вызывает C и затем C вызывает D.

   а) Если в D имеется нелокальная ссылка на переменную X из A, то которая «переменная X» будет видимой в D — из первой активации A или второй?

   б) При сделанных предположениях относительно вызовов подпрограмм и разрешения нелокальных ссылок возможно только *четыре* различных варианта статической вложенности определений подпрограмм A, B, C и D. Изобразите эти четыре варианта (например, так, как это сделано на рис. 9.4).

   в) Для каждого из этих четырех вариантов статической вложенности изобразите стек времени выполнения в тот момент, когда подпрограмма D ссылается на имя X, показав в нем записи активации для подпрограмм A, B, C и D, а также статические и динамические цепи. Для каждого случая дайте представление X в виде пары (n, m) и объясните, какой объект данных X мы должны получить с помощью нелокальной ссылки X в подпрограмме D в данной структуре времени выполнения. Предполагается, что разрешение нелокальных ссылок реализовано с использованием указателей статической цепи.
8. Предположим, что вы хотите модифицировать Pascal таким образом, чтобы разрешение нелокальных ссылок основывалось на динамической области видимости (то есть использовалось бы правило последней ассоциации), а не статической (как это происходит на самом деле).

   а) Как следовало бы модифицировать организацию центрального стека времени выполнения? Какую информацию можно было бы удалить из каждой записи активации каждой подпрограммы? Какую информацию надо было бы добавить в каждую запись активации?

б) Объясните, как в этой модифицированной реализации языка разрешались бы нелокальные ссылки.

в) Приведите два примера проверки ошибок, которые пришлось бы делать во время выполнения программы (которые могли бы быть сделаны во время компиляции, если бы использовались правила статической области видимости).

9. Предположим, что вы хотите разработать язык, в котором:

а) допускается сохранение локальных сред;

б) не допускается рекурсивный вызов подпрограмм;

в) нелокальные ссылки определяются при помощи правил динамической области видимости.

Спроектируйте реализацию такого языка. Объясните, как в нем представлена среда ссылок. Объясните, какие действия совершаются при входе и выходе из подпрограмм. Объясните, как реализовано разрешение нелокальных ссылок.

10. Объясните, почему на языке, подобном ALGOL, где параметры передаются только по значению или по имени, *невозможно* написать подпрограмму Swap с двумя параметрами, которая просто меняет местами значения этих двух параметров (параметры могут быть простыми переменными или переменными с индексами). Например, если эта подпрограмма была вызвана как Swap(X, Y), то после ее завершения переменная X должна иметь значение, равное исходному значению Y, а переменная Y — значение, равное исходному значению X. Предполагается, что подпрограмма Swap работает только с целочисленными параметрами.

11. Рассмотрим следующую программу:

```
program Main(...);
   var Y: integer;
   rocedure P(X: integer);
      begin X := X + 1; write (X, Y) end;
   begin
   Y := 1; P(Y); write(Y)
   end.
```

Напишите три числа, которые будут напечатаны в результате выполнения этой программы, если Y передается в подпрограмму P:

а) по значению;

б) по ссылке;

в) по значению-результату;

г) по имени.

12. Рассмотрим следующую программу, написанную на языке, в котором используются правила статической области видимости:

```
program main(input,output);
   var i,j,k,m: nteger;
   procedure Q(var i: integer; m: integer);
      begin
      i:=i+k;
```

```
        m:=j+1:
        writeln(i.j.k.m)
        end:
    procedure P(var i: integer: j: integer):
        var k: integer:
        begin
        k:=4:
        i:=i+k:
        j:=j+k:
        Q(i.j)
        end:
begin
i:=1:
j:=2:
k:=3:
P(i.k):
writeln(i.j.k)
end.
```

Заполните следующую таблицу для каждого оператора writeln в предположении, что передача параметров осуществляется указанным способом.

Способ передачи параметров	i	j	k	m
Язык Pascal, как записаны в программе				
Все параметры передаются по ссылке				
Все параметры передаются по значению				
Все параметры передаются по значению-результату				

13. Ссылки на массивы с использованием вычисляемых индексов (например, A[I]) иногда рассматриваются как некоторая форма создания *псевдонимов*. Например, два составных имени A[I] и A[J] можно считать псевдонимами. Объясните, почему использование таких ссылок на массивы создает для программиста и для разработчика некоторые из тех же проблем, которые возникают при использовании других типов псевдонимов.

14. В языке FORTRAN требуется, чтобы объявление для каждого объекта данных, совместно используемого через общую среду (COMMON-блок), задавалось в каждой подпрограмме, которая его использует. Это позволяет компилировать каждую подпрограмму независимо, так как в каждой из них содержатся полные объявления для всех совместно используемых объектов. Предположим, что программист случайно задал слегка разные объявления в каждой из двух подпрограмм Sub1 и Sub2, которые совместно используют переменную X. Что произойдет, если Sub1 и Sub2 будут скомпилированы по отдельности, а затем собраны в одной программе, которая будет выполнена? Положим, что во время выполнения объект данных X будет представлен в соответствии с объявлением, данным в Sub1, а не в Sub2.

15. Предположим, что подпрограмма Q (см. листинг 9.7) была бы вызвана из подпрограммы P с использованием констант в качестве фактических параметров (например, Q(2. 3) вместо Q(a. &b)). Хотя передача констант по ссылке в C запрещена, но во многих других языках это допускается, поэтому будем считать, что такая передача констант допустима и здесь. Напомним:

а) Объект данных, представляющий константу, обычно хранится в сегменте кода вызывающей подпрограммы.

б) Выражение фактического параметра, состоящее из одной константы, является частным случаем произвольного выражения, служащего фактическим параметром.

Предложите два метода реализации передачи константы: по значению и по ссылке. В первом методе не используется временное хранение объекта данных в вызывающей программе, а во втором используется. (Первый метод, использованный для передачи по ссылке, может приводить к константам, изменяющим свои значения, — например, в некоторых реализациях FORTRAN можно написать программу, которая складывает 1 и 2, получая в результате 4.) Напишите такую программу и попробуйте выполнить ее в используемой вами реализации языка. (Подсказка: используйте следующую последовательность операторов:

```
Sub(1); X := 2; X := X + 1; print(X)
```

и напишите подпрограмму Sub таким образом, чтобы она изменяла на 2 значение литерала 1.)

16. *Прием Йенсена.* Передача параметров по имени позволяет применить программистский трюк, известный как прием Йенсена. Его основная идея заключается в том, чтобы в подпрограмму передать по имени отдельными параметрами некоторое выражение, содержащее одну или более переменных, и сами эти переменные. Искусно изменяя значения этих переменных и используя ссылки на формальный параметр, соответствующий выражению, это выражение-параметр можно вычислить для многих различных значений переменных. Простой пример использования этого приема можно найти в следующей универсальной программе суммирования Sum, написанной на языке ALGOL:

```
real procedure Sum (Expr, Index, LB, UB); value LB, UB;
   real Expr; integer Index, LB, UB;
   begin real Temp; Temp := 0
   for Index := LB step 1 until UB do Temp:=Temp+Expr;
   Sum := Temp
   end Sum;
```

В этой программе переменные Expr и Index передаются по имени, а LB и UB — по значению. Результатом вызова

```
Sum(A[I],I,1,25)
```

будет сумма первых 25 элементов вектора A.

При вызове

```
Sum(A[I]*B[I],I,1,25)
```

результатом будет сумма произведений первых 25 соответствующих элементов векторов A и B (если предположить, что A и B объявлены подходящим образом).

Вызов

```
Sum(C[K,2],K,-100,100)
```

позволит получить сумму элементов второго столбца матрицы C от C[-100, 2] до C[100, 2].

а) При каком вызове Sum можно получить сумму элементов главной диагонали матрицы D, объявленной как real array D[1:50, 1:50]?

б) При каком вызове Sum можно получить сумму квадратов первых 100 нечетных чисел?

в) Используя описанный прием, напишите универсальную программу Max, которая будет возвращать максимальное из множества значений, получаемых при вычислении произвольного выражения Expr, содержащего целую переменную Index, которая изменяется в пределах от LB до UB с шагом Step.

17. В языках FORTRAN и Ada используется одинаковый синтаксис для ссылок на компоненты массива и вызова подпрограмм-функций. Например, в записи A + B(C, D) идентификатор B может означать как имя подпрограммы с двумя параметрами, так и двухмерный массив. Если объявлены имена всех массивов и подпрограмм, то такой синтаксис не приводит к неоднозначности. Но если какое-либо объявление отсутствует, то у компилятора или загрузчика могут возникнуть проблемы. При каких условиях компилятор мог бы осуществить правильную интерпретацию для пропущенного объявления?

18. Главное отличие FORTRAN 90 от FORTRAN заключается в том, что он допускает в некоторых случаях динамическое выделение памяти. Перечислите те свойства языка, для которых требуется динамическое выделение памяти транслятором.

19. Как реализовать рекурсивную функцию, например factorial(n), в языке FORTRAN 77, в котором память распределяется только статически?

20. Какие изменения следовало бы сделать в организации времени выполнения языка C, чтобы можно было добавить к нему оператор

```
int A[n]
```

который размещает в памяти массив размером n и присваивает ему имя A в предположении, что n — переменная, объявленная в другом блоке?

Можно ли сделать то же самое в Java? Почему?

21. Операция выделения памяти malloc языка C часто реализуется как обращение к программе распределения памяти, встроенной в операционную систему. Операционная система организует центральную кучу, используемую всеми программами, а операции malloc и free, вызываемые из C-программы, соответственно выделяют и освобождают память в этой центральной куче. Поскольку операция malloc требует обращения к операционной системе, этот процесс может занять много времени. Альтернативным вариантом реализации операции malloc является использование локальной кучи в адресном пространстве каждой программы. В таком случае использование этой операции обходится значительно дешевле. (Операция new в Pascal реализована таким же образом.)

Предположим, ваша программа разработана таким образом, что требует достаточно частого использования операции malloc для выделения небольших блоков памяти из кучи.

Исследуйте вашу локальную реализацию C и определите, какова в этой реализации стоимость использования для указанных целей операции `malloc`.

Сравните эту стоимость со стоимостью использования кучи и операции `malloc`, которую вы могли бы реализовать сами как часть вашей программы. Рассмотрите два случая:

а) ваша программа выделяет блоки памяти только одного размера;

б) ваша программа выделяет блоки памяти различных размеров.

22. Вернемся к программе `anomaly` из листинга 9.1. Определите, какой способ применяется в вашем локальном трансляторе Pascal для разрешения объявлений `forward`.

23. Поскольку в языке Ada допускается располагать фактические параметры в парах с соответствующими формальными, используя *имена* формальных параметров в операторе вызова, имена формальных параметров становятся *видимыми* в вызывающей программе. Обычно имена формальных параметров являются локальными для вызванной подпрограммы и не видимы для вызывающей подпрограммы. Объясните, почему подпрограмму, формальные параметры которой являются видимыми из вызывающей подпрограммы, легче модифицировать.

Глава 10. Управление памятью

В главе 9 мы обсуждали стековую структуру управления подпрограммами. Но в большинстве языков также имеется возможность динамически выделять и освобождать память для объектов данных некоторым произвольным образом. Говорят, что для таких объектов память выделяется из *кучи*. Во многих реализациях стек начинается с одного конца памяти компьютера, а куча — с другого. Если эти две структуры — стек и куча — когда-либо встречаются, то программе не хватает памяти и она останавливается. В этой главе мы обсудим различные методы управления кучей, обусловленные требованиями, предъявляемыми к языку программирования. Управление памятью для хранения данных является одной из основных забот программиста, разработчика языка и разработчика транслятора. В этой главе рассматриваются различные проблемы и методы управления памятью.

В типичных языках мы находим множество особенностей или ограничений, которые можно объяснить только желанием разработчиков применить тот или иной метод управления памятью. Рассмотрим, например, язык FORTRAN, в котором запрещаются рекурсивные вызовы подпрограмм. Рекурсивные вызовы можно было бы допустить в FORTRAN, даже не меняя его синтаксис, но реализация таких вызовов потребовала бы наличия стека точек возврата во время выполнения, а эта структура данных требует динамического управления памятью. FORTRAN, в котором запрещен рекурсивный вызов подпрограмм, может быть реализован с использованием только статического управления памятью. Огромная разница между последним стандартом этого языка, FORTRAN 90, и предыдущими версиями заключается как раз в том, что в FORTRAN 90 допускается (в ограниченном объеме) динамическое распределение памяти. Pascal был специально разработан таким образом, чтобы управление памятью основывалось на использовании стеков, LISP — чтобы допускался сбор мусора и т. д.

Несмотря на то что при разработке каждого языка обычно предполагается применение определенного метода управления памятью, детали механизмов управления, их представление в аппаратуре и программном обеспечении являются задачами разработчика транслятора языка. Например, хотя LISP позволяет использовать в качестве основы для управления памятью список свободного пространства и сбор мусора, однако существует несколько различных методов сбора мусора, из которых разработчик транслятора должен выбрать наиболее подходящий с учетом доступной аппаратуры и программного обеспечения компьютера.

К управлению памятью имеет отношение и программист, который должен при разработке программ наиболее эффективно использовать память, но у него чаще всего

почти нет прямых способов управления памятью. Программа управляет памятью только косвенно, посредством использования или неиспользования различных возможностей языка. Положение программиста еще более усложняется характерной для разработчиков языков и разработчиков трансляторов тенденцией относить управление памятью к машинно-зависимым вопросам, которые не должны явно рассматриваться в руководствах по языкам. Таким образом, программисту во многих случаях совсем не просто выяснить, какой метод управления памятью применяется на самом деле.

10.1. Размещаемые в памяти элементы

С точки зрения программиста, управление памятью — это обычно выделение областей памяти для хранения данных и кода оттранслированных программ. Однако управление памятью во время выполнения включает, помимо этих, много других областей. Некоторые из них, как, например, точки возврата из подпрограмм, нами уже обсуждались в предыдущих главах. Здесь мы рассмотрим основные элементы программ и данных, которые необходимо размещать и сохранять в памяти во время выполнения программы.

- *Сегменты кода для оттранслированных программ пользователя.* В любой системе основной блок памяти отводится для хранения сегментов кода, представляющих оттранслированную форму программ пользователя, независимо от используемого метода интерпретации — аппаратного или программного. В первом случае программы будут представлены блоками выполняемых машинных команд, во втором случае блоки будут содержать информацию в некоторой промежуточной форме.
- *Системные программы времени выполнения.* Другой значительный блок памяти во время выполнения должен быть отведен для размещения системных программ, которые поддерживают выполнение программы пользователя. Диапазон этих программ обширен — от простых *библиотечных программ*, вычисляющих тригонометрические функции или выводящих строки символов на печать, до программных интерпретаторов или трансляторов, присутствующих во время выполнения. Сюда же относятся и программы, управляющие распределением памяти во время выполнения.
- *Определяемые пользователем структуры данных и константы.* Для всех структур данных, объявленных в пользовательских программах или созданных ими, включая константы, требуется выделить место в памяти.
- *Точки возврата для подпрограмм.* Подпрограммы могут вызываться в различных точках программы. Следовательно, внутренне сгенерированная информация об управлении последовательностью действий, как то: точки возврата из подпрограмм, точки возобновления выполнения сопрограмм или сообщения о событиях для планируемых подпрограмм, — также должна быть размещена в памяти.
- *Среды ссылок.* Размещение сред ссылок (ассоциаций идентификаторов) во время выполнения программы может потребовать значительных ресурсов памяти, как, например, для А-списка в языке LISP (приложение, раздел П.6).

- *Временная память для вычисления выражений.* При вычислении выражений необходимо использовать определяемую системой временную память для хранения промежуточных результатов вычисления. Например, при вычислении выражения (x + y) × (u + v) может возникнуть необходимость сохранения результата первого сложения во временной памяти, пока будет вычисляться второе сложение. Если в выражении присутствуют рекурсивные вызовы функций, для хранения промежуточных результатов на каждом уровне рекурсии может потребоваться потенциально неограниченный объем временной памяти.
- *Временная память при передаче параметров.* При вызове подпрограммы должен быть вычислен список фактических параметров и получаемые результаты должны сохраняться во временной памяти, пока не будет полностью вычислен весь список параметров. Если вычисление какого-либо параметра приводит к рекурсивным вызовам функции, то, как и в случае вычисления выражений, может потребоваться потенциально неограниченный объем памяти.
- *Буферы ввода-вывода.* Обычно операции ввода и вывода работают с использованием буферов, которые служат в качестве временных областей памяти для хранения данных в промежутке между моментом их фактической передачи с внешнего устройства или на него и моментом выполнения инициированных программой операций ввода или вывода.
- *Различные системные данные.* В реализации почти каждого языка требуется память для хранения различных системных данных: таблиц, информации о состоянии устройств ввода-вывода и другой разнообразной информации о состоянии (например, счетчики ссылок или биты сбора мусора).

Кроме перечисленных элементов программ и структур данных, некоторые операции также требуют выделения или освобождения памяти. Ниже перечислены наиболее существенные из таких операций.

- *Операции вызова и возврата из подпрограмм.* При вызове подпрограммы требуется выделить память для размещения записи активации подпрограммы, среды локальных ссылок и других данных. При выполнении операции выхода из подпрограммы обычно требуется освободить дополнительную память, выделенную при вызове подпрограммы.
- *Операции создания и уничтожения структур данных.* Если в языке предусмотрены операции, позволяющие в произвольной точке программы (а не только на входе в подпрограмму) создавать новые структуры данных (например, операция new в языке Java), то эти операции обычно требуют выделения памяти отдельно от той, которая выделяется при входе в подпрограмму. Язык может также предоставлять явные операции уничтожения ранее созданных структур данных, например функции dispose в Pascal и free в C, которые могут потребоваться для освобождения выделенной под какие-то структуры данных памяти. В Java такая явная операция отсутствует; для этого предназначен процесс сбора мусора.
- *Операции вставки и удаления компонентов.* Если язык предоставляет операции вставки и удаления компонентов структур данных, то для их реализа-

ции может потребоваться выделение и освобождение памяти (например, функция push в языке Perl добавляет элемент к массиву).

Хотя перечисленные операции требуют явного управления распределением памяти, существует множество других операций, которые подразумевают некоторое скрытое управление памятью, по большей части связанное с выделением и освобождением памяти при осуществлении различных вспомогательных действий (например, при вычислении выражений и передаче параметров).

10.2. Память, управляемая программистом и системой

До какой степени следует позволять программисту непосредственно управлять распределением памяти? С одной стороны, популярность языка C объясняется отчасти тем, что он предоставляет программисту широкие возможности явного управления памятью посредством операций malloc и free, которые соответственно выделяют память для структур данных, определяемых программистом, и освобождают ее. С другой стороны, многие языки высокого уровня не дают программисту никаких средств прямого управления памятью; на распределение ресурсов памяти можно влиять только косвенным образом путем использования тех или иных возможностей языка.

Возможность прямого управления распределением памяти программистом связана с двумя трудностями: во-первых, это управление может оказаться тяжелым и часто нежелательным бременем для программиста, и, во-вторых, оно может вступать в противоречие с осуществляемым системой необходимым управлением распределением памяти. Ни один язык высокого уровня не может позволить программисту брать на себя все заботы по распределению памяти. Например, едва ли можно ожидать, что программист будет заниматься выделением временной памяти для хранения промежуточных результатов вычислений, точек возврата из подпрограмм или других системных данных. В лучшем случае программист мог бы управлять распределением памяти для локальных данных (и, возможно, программ). Но даже простое размещение в памяти структур данных или освобождение памяти, допустимые в C, с большой вероятностью ведут к появлению мусора и повисших ссылок. Таким образом, управление распределением памяти программистом представляет опасность и для самого программиста, так как может приводить к тонким ошибкам или потере доступа к свободным областям памяти. Распределение памяти, осуществляемое программистом, также может вмешиваться в распределение памяти, осуществляемое системой, по той причине, что могут потребоваться специальные области памяти и специальные программы управления ею, приводящие к менее эффективному использованию памяти в целом.

Преимущество возможности управления распределением памяти программистом заключается в том, что часто системе бывает исключительно трудно определить, когда наиболее целесообразно занимать и освобождать память. Программист же часто знает достаточно точно, когда необходима конкретная структура данных, а когда она больше не нужна и занимаемая ею память может быть освобождена.

Эта дилемма ни в коем случае не может считаться тривиальной и часто является центральной при выборе языка для реализации определенного проекта. Следует ли выбрать строго типизированный язык с эффективной системой управления памятью, который обеспечивает надежность выполнения программ с соответствующим уменьшением скорости выполнения, или следует предпочесть язык, позволяющий ускорить выполнение программ (например, за счет увеличения скорости вычисления и распределения ресурсов памяти) с соответствующим увеличением вероятности возникновения ошибок, зависания и даже непредвиденных остановок программы? Это один из фундаментальных вопросов, обсуждаемых в среде специалистов в области технологии программирования. В данной книге нет решения этой проблемы, наша основная цель — предоставить читателю соответствующую подробную информацию для того, чтобы он мог самостоятельно принимать решение в каждом конкретном случае.

Фазы управления памятью

Принято выделять три основных аспекта управления памятью.

1. *Начальное распределение.* В момент начала выполнения программы каждый блок памяти может быть либо занят, либо свободен. Если первоначально блок памяти свободен, то он доступен для динамического распределения в процессе выполнения программы. Любая система управления памятью требует наличия какого-либо метода для отслеживания и учета свободной памяти, а также механизма, позволяющего распределять эту память по мере необходимости в процессе выполнения программы.
2. *Восстановление памяти.* Память, которая была распределена и использовалась в течение какого-то времени, после того как занимающий ее объект перестал быть нужным, должна быть восстановлена для повторного использования. Восстановление памяти может быть очень простым, как в случае перемещения указателя стека, или очень сложным, как в случае сбора мусора.
3. *Уплотнение и повторное использование.* Восстановленная память может быть сразу же повторно использована, но может потребоваться и процедура ее уплотнения — построение более крупных блоков свободной памяти из маленьких. Повторное использование памяти обычно подразумевает применение тех же механизмов, что и при начальном распределении.

В настоящее время известны и используются в реализациях языков много различных методов управления памятью. В данной книге невозможно рассказать обо всех таких методах, но для того, чтобы получить представление об основных принципах, достаточно познакомиться с отдельными примерами. Большинство используемых методов являются всего лишь вариациями какого-либо из рассматриваемых ниже основных методов.

10.3. Статическое управление памятью

Простейший способ распределения памяти — это *статическое распределение*, то есть распределение во время трансляции, которое остается неизменным во время

выполнения. Обычно для сегментов кода пользовательских и системных программ, а также для буферов ввода-вывода и различных системных данных память распределяется статически. Такое распределение памяти не требует выполнения каких-либо действий по управлению памятью во время выполнения программы, и конечно, в таком случае вопросы восстановления и повторного использования памяти отпадают.

В обычной реализации FORTRAN вся память распределяется статически. Каждая подпрограмма компилируется отдельно, при этом компилятор создает сегмент кода (включая запись активации), содержащий скомпилированную программу, ее области данных, временные переменные, точку возврата, а также различные системные данные. Загрузчик выделяет пространство в памяти для этих скомпилированных блоков во время загрузки, так же как и для системных программ, необходимых во время выполнения. Во время выполнения программы не требуется никакого управления памятью.

Статическое распределение памяти эффективно, так как не требует дополнительных временных и пространственных затрат на управление памятью в процессе выполнения. Транслятор может непосредственно сгенерировать *l*-значения (адреса) для всех элементов данных. Однако статический метод несовместим с рекурсивным вызовом подпрограмм, со структурами данных, чьи размеры зависят от входных или вычисляемых в программе данных, а также со многими другими желательными возможностями языка. В следующих разделах этой главы мы обсудим различные методы *динамического* (во время выполнения) *управления памятью*. Читатель тем не менее не должен упускать из виду важность статического распределения памяти. Два широко используемых языка программирования, FORTRAN и COBOL, разработаны в расчете на исключительно статическое распределение памяти (хотя, как было уже сказано, теперь FORTRAN 90 позволяет использовать динамические массивы и рекурсивные процедуры). В языках, подобных С, которые используют динамическое распределение памяти, допускается также и статическое управление для повышения эффективности.

10.4. Управление кучей

Третий основной вид управления памятью после стекового и статического называется *управлением кучей*. *Куча* — это блок памяти, области которого выделяются и освобождаются некоторым относительно произвольным образом. В этом случае проблемы выделения, восстановления, уплотнения и повторного использования памяти могут стать весьма серьезными. Не существует какого-либо одного метода управления кучей; скорее, следует говорить о наборе методов для регулирования различных аспектов управления при такой организации памяти.

Необходимость в куче возникает тогда, когда язык позволяет распределять и освобождать память в произвольных точках выполнения программы, а также тогда, когда язык предоставляет средства для создания, разрушения и расширения пользовательских структур данных в произвольных местах программы. Например, в ML в любой момент можно объединить два списка в один или динамически определить новый тип данных. В LISP в любой точке программы к существующему

списку может быть добавлен новый элемент, что опять-таки требует выделения дополнительной памяти. В обоих этих языках возможно также освобождение памяти в непредсказуемые моменты во время выполнения.

Методы управления кучей удобно разделить на две категории в зависимости от того, имеют ли размещаемые в куче элементы один и тот же фиксированный размер или их размер может меняться. В первом случае методы управления можно значительно упростить. Уплотнение, в частности, перестает быть проблемой, так как все доступные элементы имеют один и тот же размер. В этом разделе мы рассмотрим случай с элементами фиксированного размера, а более сложный случай с элементами переменного размера рассмотрим в следующем разделе.

10.4.1. Обзор языка LISP

История. Впервые язык LISP был разработан и реализован группой программистов под руководством Джона Мак-Карти (John McCarthy) из Массачусетсского технологического института (МТИ) примерно в 1960 г. [80]. Этот язык широко использовался для научных исследований, в основном в области искусственного интеллекта (робототехника, обработка естественных языков, доказательство теорем, системы искусственного интеллекта). За последние тридцать лет было разработано множество версий языка LISP. Из всех описанных в этой книге языков только LISP не стандартизирован и не имеет какой-либо одной доминирующей реализации.

LISP во многих аспектах отличается от большинства других языков. Наиболее удивительная особенность LISP — это эквивалентность форм представления в языке программ и данных, что позволяет выполнять структуры данных как программы и модифицировать программы как данные. Другой отличительной особенностью является применение в качестве основной управляющей структуры рекурсии, а не итерации (циклов), как в большинстве других языков программирования.

Как сказано ранее, LISP зародился в МТИ приблизительно в 1960 г. В 60-е и 70-е гг. появилось множество версий этого языка. В числе прочих существовали MacLisp в МТИ, Interlisp для DEC PDP-10, разработанный Уорреном Тейтелманом (Warren Teitelman), а также были Spice LISP и Franz LISP. Основной доминирующей версией в это время был Interlisp. Во второй половине 70-х гг. Джеральд Сьюсманн (Gerald Susmann) и Гай Стил (Guy Steel) в процессе исследования моделей вычисления разработали очередную версию языка, получившую название Schemer. Необходимость ограничения длины имени шестью символами превратила этот язык в Scheme. Именно эта версия LISP оказала наибольшее влияние на использование языка в академической среде.

В апреле 1981 г. была организована встреча заинтересованных лиц, имеющих отношение к LISP, для объединения различных диалектов в единый язык, который за неимением лучшего названия стал называться Common LISP (общий LISP). Название Standard LISP (стандартный LISP) было уже использовано для одного из диалектов. Разработка основной структуры Common LISP заняла следующие три года.

Пик популярности языка LISP приходится на период с 1985 по 1990 г. В это время задачи искусственного интеллекта вызывали повышенный интерес; более

75 % первокурсников выбирали искусственный интеллект в качестве области специализации. В 1986 г. техническая рабочая группа X3J13 начала работу по стандартизации Common LISP, и ее первой целью стало объединение Common LISP и Scheme. Однако эта попытка не увенчалась успехом, поэтому в 1989 г. в Институте инженеров электротехники и электроники (Institute of Electrical and Electronics Engineers, IEEE) был разработан стандарт IEEE языка Scheme. Приблизительно в это же время стали заметны преимущества объектно-ориентированного программирования на языках C++ и Smalltalk, поэтому был разработан Common LISP Object System (CLOS). Наконец, в 1992 г. группа X3J13 предложила описание Common LISP, который занял более тысячи страниц — больше, чем описание стандарта языка COBOL. Кажется странным, что язык с такой простой и ясной основной структурой вышел из-под контроля.

С самого начала LISP подвергался критике за медленность выполнения, особенно на стандартном компьютере фон Неймана, описанном в главе 2. Когда функции car и cdr были смоделированы на аппаратном компьютере IBM 704, была предложена альтернативная машинная архитектура, которая позволяла ускорить выполнение программ на LISP. Несколько компаний разрабатывали компьютеры, специально предназначенные для быстрого выполнения программ на LISP. Тем не менее около 1989 г. была разработана стратегия сбора мусора для компьютера со стандартной неймановской архитектурой, которая оказалась конкурентоспособной по отношению к специально созданному для выполнения программ на LISP аппаратному компьютеру. По этой причине ни одна из компаний, создававших специальные LISP-компьютеры, не имела долговременного коммерческого успеха.

Краткий обзор языка. Обычно программы на LISP выполняются в интерактивной среде. Поэтому не существует главной программы в обычном понимании этого слова. Вместо этого пользователь вводит с терминала последовательность выражений, которые требуется вычислить. Система LISP вычисляет выражения по мере их ввода и автоматически выводит результаты на экран монитора. Обычно некоторые введенные выражения являются определениями функций. Другие выражения содержат обращения к этим определенным функциям с указанием конкретных значений аргументов. В LISP не существует блочной структуры или других синтаксически сложных структур. Единственной формой взаимодействия между различными функциями является обращение к одной функции из другой во время выполнения программы.

Функции LISP полностью определяются как выражения. Каждый оператор является функцией, возвращающей некоторое значение, а подпрограммы записываются как единые (часто очень сложные) выражения. Для того чтобы этот синтаксис, включающий только выражения, сделать более похожим на обычный синтаксис, состоящий из *последовательности операторов*, в язык были добавлены разнообразные специальные конструкции, но все же основной формой остаются выражения.

Набор типов данных в LISP довольно ограничен. Основными элементарными типами данных являются *литеральные атомы* (символы) и *числовые атомы* (числа). Связанные списки и списки свойств (особая разновидность связанных списков) образуют основные структуры данных. Вся обработка дескрипторов осуществляется во время выполнения программы, и не требуется никаких объявлений.

LISP предоставляет широкий выбор элементарных операций для создания, уничтожения и модификации списков (включая списки свойств). Имеются и основные арифметические операции. Во время выполнения программы с помощью специальных элементарных операций можно транслировать и выполнять другие программы, а программы могут создаваться и выполняться динамически.

Структуры управления LISP относительно просты. Выражения, используемые для составления программ, записываются строго в кембриджской польской записи и могут содержать условное ветвление. Конструкция prog предоставляет простую структуру для записи последовательности выражений. В большинстве программ на LISP широко используются рекурсивные вызовы функций.

Комментарии в LISP обычно начинаются точкой с запятой, при этом вся следующая за этим символом строка является комментарием.

Обработка нелокальных ссылок в LISP основана на *правиле последней ассоциации*, которое часто реализуется с помощью простого связанного списка текущих ассоциаций, так называемого *А-списка*. Каждый раз, когда встречается ссылка на какой-либо идентификатор, в этом списке для него отыскивается текущая ассоциация.

Параметры функций передаются либо только по значению, либо только по имени в зависимости от классификации функции, при этом обычным случаем является передача параметров по значению.

Легче всего LISP реализуется с помощью программного интерпретатора и программного моделирования всех элементарных операций. Многие реализации предоставляют также компилятор, который можно использовать для компиляции отдельных определений функций в машинные коды. Эти скомпилированные функции затем выполнятся аппаратным интерпретатором (но по-прежнему для многих операций требуется программное моделирование). LISP довольно плохо приспособлен для компиляции, так как большая часть связываний происходит только во время выполнения программы. В качестве основного средства для хранения данных и программ применяется сложная система управления памятью с использованием кучи и сбора мусора.

10.4.2. Элементы фиксированного размера

Предположим, что каждый элемент фиксированного размера, расположенный в куче и впоследствии извлекаемый из нее, занимает N слов памяти. Обычно N равно 1 или 2. Если предположить, что куча занимает непрерывный блок памяти, мы мысленно разбиваем блок на последовательность K элементов длиной по N слов каждый, так что размер всей кучи равен $K \times N$. Когда требуется некоторый элемент, выделяется один из этих элементов кучи. Когда некоторый элемент освобождается, он должен быть одним из этих же исходных элементов кучи.

Первоначально эти K элементов связаны друг с другом и образуют *список свободного пространства* (то есть первое слово каждого элемента списка свободного пространства указывает на первое слово следующего элемента списка свободного пространства). При выделении элемента из списка свободного пространства удаляется его первый элемент, и указатель на него возвращается операции, требующей выделения памяти. Когда элемент освобождается, он просто вставляется сно-

ва в голову списка свободного пространства. На рис. 10.1 изображены исходный вид списка свободного пространства и его вид после того, как были выделены, а затем освобождены некоторые элементы.

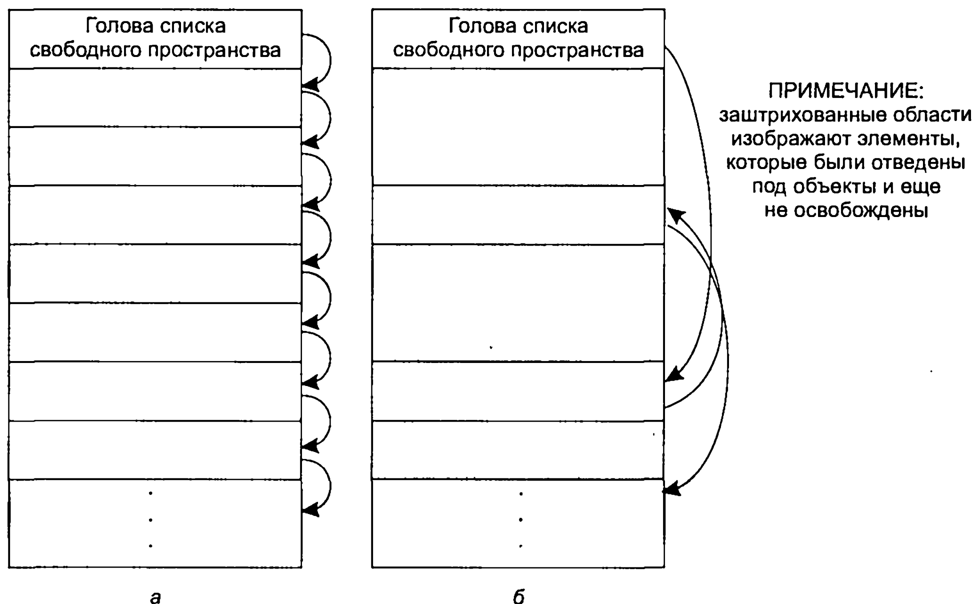


Рис. 10.1. Структура списка свободного пространства: *а* — начальный список свободного пространства; *б* — список свободного пространства после выполнения

Восстановление памяти: счетчики ссылок и сбор мусора

Возврат освободившейся памяти в список свободного пространства несложен при условии, что ее можно обнаружить и восстановить. Однако процесс обнаружения и восстановления освободившейся памяти может оказаться достаточно сложным. Проблема заключается в том, чтобы определить, какие элементы в куче могут быть повторно использованы и, следовательно, должны быть возвращены в список свободного пространства. Широко применяются три способа решения этой задачи, которые описаны ниже.

Явный возврат программистом или системой. Самый простой метод восстановления — это *явный возврат*. Когда элемент, бывший в употреблении, становится доступным для повторного использования, он должен быть явным образом обозначен как свободный и возвращен в список свободного пространства (в Pascal, например, для этого нужно вызвать функцию dispose). Если элементы использовались для системных целей, например для хранения сред ссылок, точек возврата из подпрограмм или временных переменных, или если все управление памятью контролируется операционной системой, то каждая системная программа обеспечивает возвращение пространства памяти, чтобы оно стало доступным для повторного использования, путем явного обращения к программе, выполняющей освобождение памяти, передавая ей в качестве параметра соответствующий элемент.

Явный возврат — вполне естественный способ восстановления памяти при ее организации в виде кучи, но, к сожалению, не всегда его можно применить. Причиной этого являются две старые проблемы: мусор и повисшие ссылки. Мы уже обсуждали эти проблемы в главе 5 в связи с разрушением структур данных. Если структура разрушена (и занимаемая ее память освободилась) до разрушения всех путей доступа к этой структуре, то все оставшиеся пути доступа становятся повисшими ссылками. С другой стороны, если последний путь доступа к структуре разрушен без разрушения самой структуры и восстановления памяти, то эта структура становится мусором. С точки зрения управления кучей повисшая ссылка — это указатель на элемент, который был возвращен в список свободного пространства (а затем мог быть повторно использован для какой-либо иной цели). Мусором с этой точки зрения является элемент, который доступен для повторного использования, но отсутствует в списке свободного пространства и, следовательно, недоступен.

Если происходит накопление мусора, то объем доступной памяти постепенно уменьшается, в результате чего выполнение программы может прекратиться за неимением доступного свободного пространства памяти. При попытке программы с помощью повисшей ссылки изменить структуру данных, отведенная память под которую уже была сделана свободной, может случайно измениться содержимое элемента из списка свободного пространства. Если в результате этого изменится указатель, связывающий этот элемент со следующим элементом списка свободного пространства, весь остаток списка может исказиться. Хуже того, последующие попытки программы, отвечающей за распределение памяти, использовать указатель в элементе такого измененного списка могут привести к совершенно непредсказуемым результатам (например, в качестве свободного пространства может быть выделена и позднее изменена область внутри выполняемого кода). Аналогичные проблемы возникают в случае, если элемент, на который указывает повисшая ссылка, был уже перераспределен для других целей.

Явный способ восстановления памяти, организованной в виде кучи, предрасполагает к появлению мусора и повисших ссылок, так как программист очень легко может сделать ошибку, приводящую к этим последствиям. Рассмотрим, например, следующие операторы С:

```
int *p, *q;                   /* p и q являются указателями на целые переменные */
...
p = malloc(sizeof(int));      /* выделить память для int и присвоить ее
                                 l-значение переменной p */
p = q;                        /* l-значение, содержавшееся в переменной p,
                                 потеряно */
```

или

```
int *p, *q;                   /* p и q являются указателями на целые числа */
...
p = malloc(sizeof(int));      /* выделить память для int и присвоить ее
                                 l-значение переменной p */
q = p;                        /* сохранить l-значение выделенной памяти
                                 в переменной q */
free(p);                      /* образуется повисшая ссылка в переменной q */
```

Системе, функционирующей во время выполнения, одинаково трудно избежать создания мусора и повисших ссылок. В LISP, например, связанные спис-

ки — это основная структура данных. Одна из элементарных операций LISP, cdr, получает указатель на некоторый элемент связанного списка и возвращает в качестве результата указатель на следующий элемент этого списка (рис. 10.2, *а*). Элемент, указанный в операции cdr, *может* быть освобожден ею при условии, что исходный указатель, переданный в cdr, является единственным указателем на этот элемент. Если при этом операция cdr не возвратит элемент в список свободного пространства, то элемент станет мусором. С другой стороны, если cdr возвратит элемент в список свободного пространства, в то время как на него имеются другие указатели, то эти указатели становятся повисшими ссылками. Если отсутствует непосредственный способ определения наличия таких указателей, то операция cdr потенциально должна порождать мусор или повисшие ссылки.

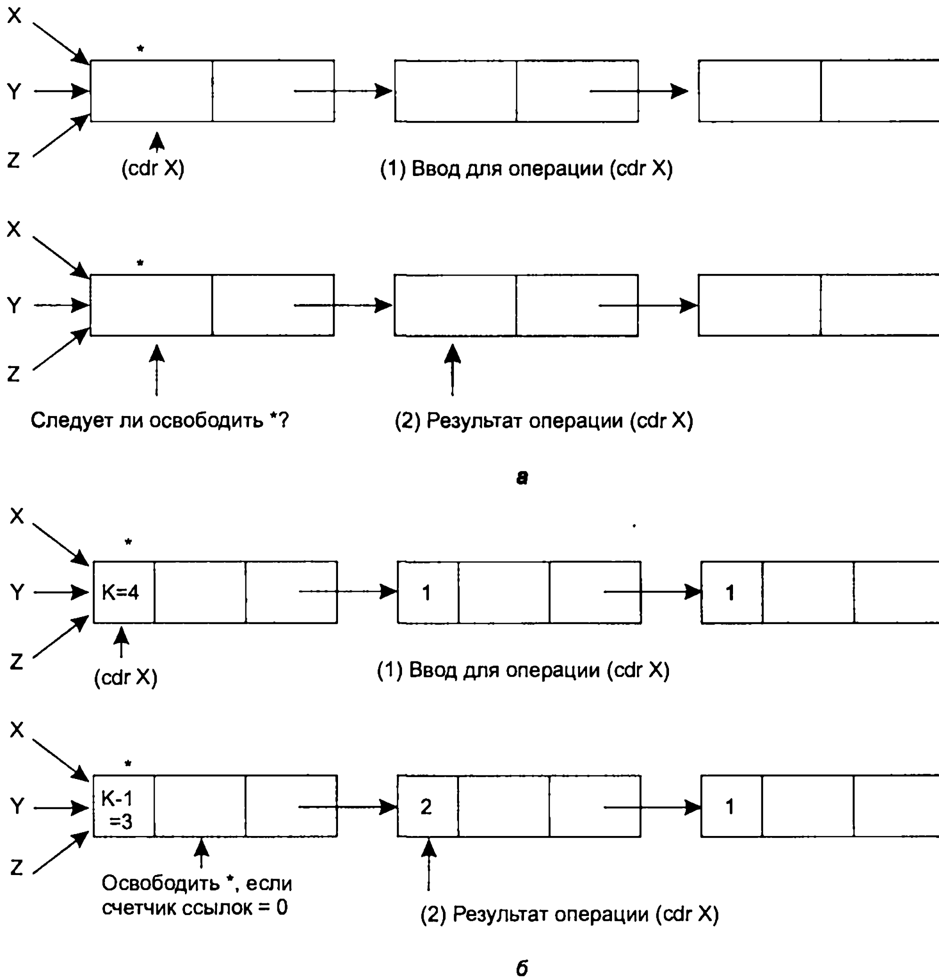


Рис. 10.2. Операция cdr языка LISP: а — модель сбора мусора; б — модель счетчика ссылок

Поскольку явный возврат связан с описанными проблемами, желательно иметь альтернативные способы восстановления памяти. Одним из вариантов являются *счетчики ссылок*, которые также требуют явного возврата, но обеспечивают возможность проверять количество указателей на данный элемент, так чтобы не создавались повисшие ссылки. Другой альтернативой является *сбор мусора*, где допускается возникновение мусора, но не повисших ссылок. Когда список свободного пространства исчерпывается, активизируется механизм сборки мусора, который находит мусор и восстанавливает его, возвращая в список свободного пространства.

Счетчики ссылок. Использование счетчиков ссылок — более простой из двух упомянутых методов. Внутри каждого элемента кучи выделяется некоторое дополнительное пространство для *счетчика ссылок*, который содержит целое число, равное количеству ссылок на этот элемент в данный момент времени. Когда этот элемент отводится под какой-либо объект (то есть когда он выделяется из списка свободного пространства), счетчик ссылок устанавливается равным 1. Каждый раз, когда на этот элемент создается новый указатель, значение счетчика ссылок увеличивается на 1. Каждый раз, когда разрушается указатель на этот элемент, значение счетчика ссылок уменьшается на 1. Когда счетчик ссылок достигает значения 0, элемент освобождается и может быть возвращен в список свободного пространства.

Счетчики ссылок в большинстве ситуаций позволяют избежать как появления повисших ссылок, так и накопления мусора. Снова рассмотрим операцию cdr языка LISP. Если каждый элемент списка содержит счетчик ссылок, операции cdr будет несложно избежать описанные выше трудности. cdr вычтет 1 из счетчика ссылок элемента, на который первоначально указывал ее аргумент. Если в результате получается, что счетчик ссылок становится равным 0, то элемент можно возвратить в список свободного пространства, если же счетчик ссылок не равен нулю, то на этот элемент все еще указывают какие-то другие указатели и его нельзя считать свободным (рис. 10.2, *б*).

Когда программист имеет возможность использовать явный оператор *освобождения* или *удаления* структуры, то счетчики ссылок также обеспечивают защиту. Результатом выполнения оператора *освобождения* будет уменьшение на 1 счетчика ссылок данной структуры. Действительный возврат этой структуры в список свободного пространства производится, когда ее счетчик станет нулевым. Ненулевой счетчик указывает, что структура все еще доступна и оператор *освобождения* должен игнорироваться.

Самый большой недостаток счетчиков ссылок — это стоимость их использования. Во время работы программы непрерывно должны происходить проверки счетчиков ссылок, их уменьшение и увеличение, в результате чего заметно падает эффективность работы программы. Рассмотрим, к примеру, простой оператор присваивания P := Q, где P и Q являются переменными-указателями. Без использования счетчиков ссылок скопировать значение указателя Q в P достаточно просто. Если задействованы счетчики ссылок, приходится делать следующее.

1. Обратиться к элементу, на который указывает P, и уменьшить его счетчик ссылок на 1.
2. Проверить получившееся значение счетчика ссылок и, если оно равно 0, вернуть этот элемент в список свободного пространства.

3. Скопировать *l*-значение указателя из Q в P.
4. Обратиться к элементу, на который указывает Q, и увеличить его счетчик ссылок на 1.

Общая стоимость операции присваивания значительно возрастает. Любая подобная операция, которая создает или разрушает указатели, неизбежно связана с изменением счетчиков ссылок. Помимо того, следует учитывать стоимость размещения этих счетчиков в памяти. Если первоначально элементы занимали один или два слова, то счетчики ссылок могут значительно уменьшить объем памяти, доступный для размещения данных. Тем не менее эта технология пользуется успехом в системах параллельной обработки, в которых стоимость поддержания счетчиков распределяется между различными пользователями данных, в то время как в описанной ниже системе сбора мусора получение данных обходится достаточно дорого.

Сбор мусора. Возвращаясь к основным проблемам мусора и повисших ссылок, можно с уверенностью сказать, что повисшие ссылки потенциально намного опаснее мусора. Накопление мусора может привести к истощению свободной памяти, но повисшие ссылки могут привести к полному хаосу вследствие случайных изменений используемой памяти. Разумеется, эти две проблемы взаимосвязаны: повисшие ссылки возникают, если память освобождается слишком рано, а мусор — когда память, наоборот, слишком долго не освобождается. В ситуации, когда нереально или слишком дорого избежать одновременного возникновения обеих проблем с помощью механизма, подобного счетчикам ссылок, лучше предпочесть наличие в памяти мусора, но избежать повисших ссылок. Лучше вообще не восстанавливать память, чем сделать это слишком рано.

Основная концепция сбора мусора заключается в том, что лучше допустить накопление мусора, зато избежать возникновения повисших ссылок. Когда список свободного пространства полностью исчерпан и требуются новые ресурсы памяти, то выполнение программы временно приостанавливается и активизируется специальная процедура сбора мусора, которая обнаруживает элементы кучи, ставшие мусором, и возвращает их в список свободного пространства. Затем возобновляется выполнение программы с того места, на котором оно было прервано, и вновь начинается накопление мусора, пока не исчерпается список свободного пространства. Тогда снова будет инициирована процедура сбора мусора и т. д.

Поскольку сбор мусора происходит довольно редко (а именно, когда исчерпывается список свободного пространства), то этой процедуре позволительно быть дорогостоящей. Она включает две стадии.

1. *Маркировка элементов.* На первой стадии должен быть помечен каждый активный элемент кучи (то есть тот, который является частью доступной структуры данных). Каждый элемент должен содержать *бит сбора мусора*, который вначале устанавливается в положение «включен». Алгоритм маркировки устанавливает бит сбора мусора каждого активного элемента в положение «выключен».

2. *Сбор элементов.* После того как алгоритм маркировки отметил активные элементы, все остальные элементы, у которых бит сбора мусора установлен в положении «включен», рассматриваются как мусор и могут быть возвращены в список свободного пространства. Для этого достаточно просто по-

следовательно просмотреть всю кучу. По мере просмотра проверяется бит сбора мусора каждого элемента. Если этот бит находится в положении «выключен», то элемент пропускается; если же в положении «включен», то элемент присоединяется к списку свободного пространства. Во время этого просмотра все биты сбора мусора устанавливаются в положение «включен» (для подготовки к последующему сбору мусора).

Наиболее сложной частью процедуры сбора мусора является маркировка. Поскольку в момент инициирования процедуры сбора мусора ресурсы памяти исчерпаны, любой элемент в куче либо активен (то есть используется), либо является мусором. К сожалению, анализ самого элемента не позволяет определить, является ли он мусором, поскольку ничто внутри элемента не указывает на то, что он больше не доступен из другого активного элемента. Более того, даже наличие указателя на этот элемент из какого-либо другого элемента кучи не гарантирует, что этот элемент является активным; дело в том, что оба элемента могут оказаться мусором. Следовательно, простого сканирования кучи, при котором просматриваются указатели и элементы, на которые существуют указатели, помечаются как активные, недостаточно.

Когда элемент кучи является активным? Очевидно, в том случае, если на этот элемент есть указатель из другого *активного* элемента кучи или *извне* кучи. Если можно определить все такие внешние указатели и отметить соответствующие элементы кучи, то можно начинать итерационный процесс маркировки, в ходе которого ищутся в этих активных элементах указатели на другие, пока не отмеченные элементы. Эти новые элементы также отмечаются и в них ищутся новые указатели и т. д. Требуется достаточно аккуратно использовать указатели, поскольку при маркировке подразумевается выполнение следующих трех условий:

1) любой активный элемент должен быть достижим по цепи указателей, начинающейся вне кучи;
2) должна иметься возможность обнаружения всех указателей вне кучи, указывающих на элемент внутри нее;
3) для каждого активного элемента должна иметься возможность определить все поля внутри него, которые содержат указатели на другие элементы кучи.

В LISP эти три условия выполняются, что позволяет использовать в этом языке технологию сбора мусора. Но если хотя бы одно их этих предположений не выполнено, то в процессе маркировки некоторые активные элементы могут быть пропущены. Например, в C предположения 2 и 3 могут не выполняться. Если бы в C использовался сбор мусора, в результате могли бы быть восстановлены активные элементы и, таким образом, образовались бы повисшие ссылки.

Полезно разобрать, каким образом достигается выполнение этих предположений в типичной реализации LISP. Во-первых, все элементы кучи имеют одинаковый формат, обычно состоящий из двух полей для указателей и ряда дополнительных битов для системных данных (в том числе бита сбора мусора). Поскольку каждый элемент кучи содержит ровно два указателя и они расположены всегда в одних и тех же позициях в элементе, предположение 3 всегда выполняется. Во-вторых, имеется только небольшой набор системных структур данных, которые могут содержать указатели на элементы кучи. Если начинать маркировку с этих

системных структур данных, можно с уверенностью сказать, что все указатели внутрь кучи будут обнаружены, тем самым гарантируется выполнение условия 2. Наконец, к элементу кучи можно получить доступ только по цепи указателей, начинающейся вне кучи. Например, указатель на некоторый элемент кучи не может быть получен добавлением константы к какому-либо другому указателю — таким образом, выполняется условие 1. В примере 10.1 показано, как во время выполнения программы на LISP процедура сбора мусора взаимодействует со стеком и кучей.

Пример 10.1. Распределение памяти в LISP

Взаимоотношения между стеком и кучей в LISP проиллюстрированы на рис. 10.3. На этом рисунке предполагается следующее:

- куча содержит 15 элементов, из которых 9 в настоящий момент принадлежат списку *free* свободного пространства (см. рис. 10.3, *а*);
- пользователем были введены следующие два определения:

```
(defun fl(x y z) (cons x (f2 y z)))
(defun f2(v w) (cons v w))
```

Выполнение выражения `(fl 'a '(b c) '(d e))` происходит следующим образом:

1. Вызывается функция `fl` и аргументы `x`, `y` и `z` добавляются в стек с использованием девяти доступных элементов кучи из списка `free` свободного пространства (см. рис. 10.3, *б*).
2. Вызывается функция `f2` с указателями на ее аргументы `v` и `w` (см. рис. 10.3, *в*).
3. Список `free` свободного пространства пуст. В процессе сборки мусора сначала отмечаются элементы, на которые имеются указатели из стека, а на втором проходе все оставшиеся элементы помещаются в список `free` свободного пространства (см. рис. 10.3, *г*).
4. Вычисляется значение для `f2` и помещается в стек (см. рис. 10.3, *д*).
5. Вычисляется значение для `fl` и помещается в стек (см. рис. 10.3, е). Система LISP автоматически отображает результат для пользователя.
6. Все элементы в вычислении теперь являются мусором. Когда список `free` свободного пространства в очередной раз станет пустым, эти элементы будут восстановлены для повторного использования во время очередной инициализации процедуры сбора мусора.

10.4.3. Элементы переменного размера

Управление кучей, в которой программист выделяет и возвращает элементы переменного размера, сложнее, чем в случае с использованием элементов фиксированного размера, хотя многие из уже описанных концепций применимы и здесь. Элементы переменного размера встречаются во многих ситуациях. Например, если память используется для определяемых программистом структур данных, сохраняемых последовательно, например массивов, то требуются блоки памяти различных размеров; то же самое касается и записей активаций для задач, которые могут быть размещены в куче в последовательных блоках переменных размеров.

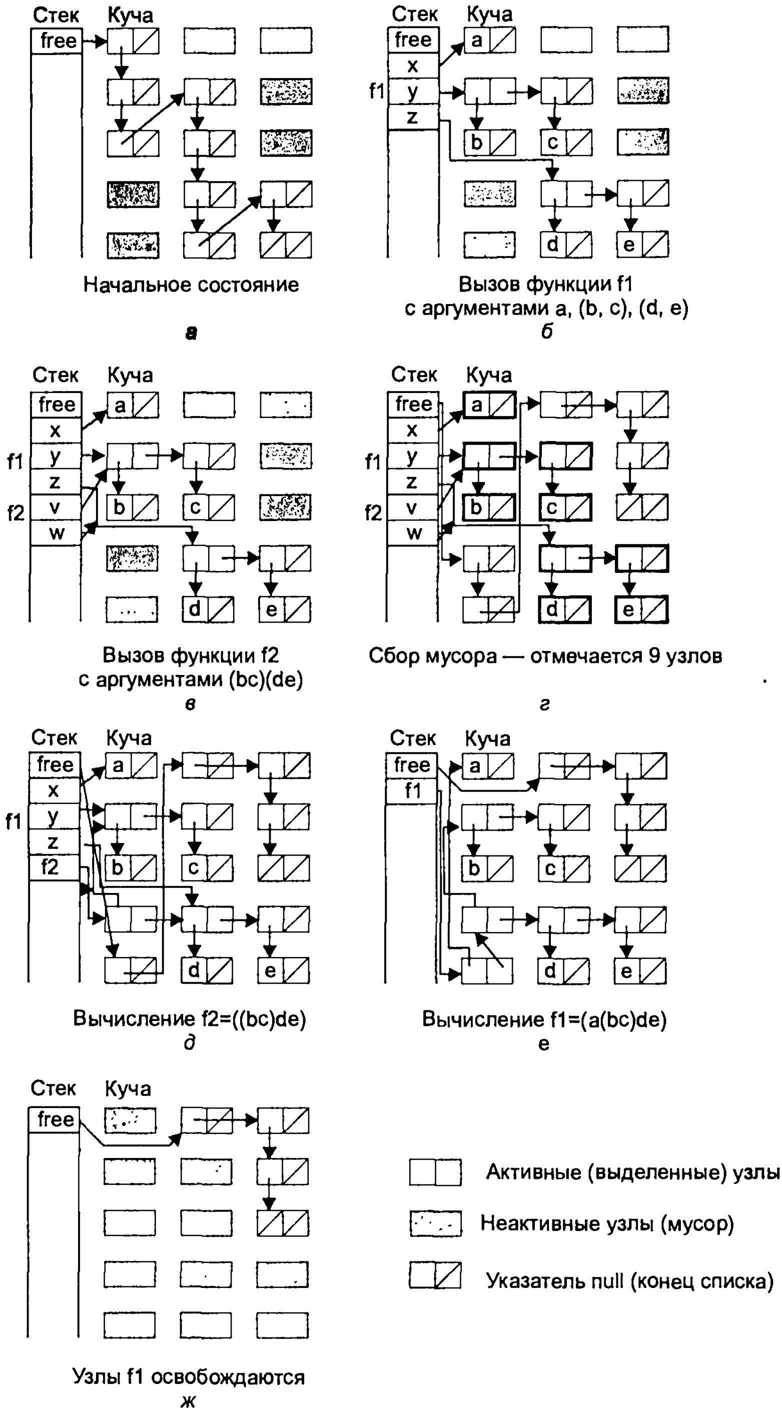


Рис. 10.3. Расположение в памяти кучи и стека языка LISP

Основные трудности в случае элементов переменного размера связаны с повторным использованием освободившейся памяти. Даже если мы возвращаем в список свободных элементов кучи два блока размером по пять слов каждый, может оказаться невозможным удовлетворить последующий запрос на блок из шести слов. Такая проблема не возникала в более простом случае использования блоков фиксированного размера, возвращаемое пространство всегда могло быть сразу же повторно использовано.

Начальное распределение и повторное использование

В случае использования элементов фиксированного размера можно было сразу же разбить кучу на множество элементов и затем осуществить начальное распределение памяти на основе списка свободного пространства, содержащего эти элементы. Но для элементов переменного размера такой метод неприменим. Вместо этого нам желательно поддерживать свободное пространство в блоках максимально возможного размера. Тогда мы изначально рассматриваем всю кучу как просто один блок свободной памяти. Для начального распределения подходит *указатель кучи*. Когда запрашивается блок из N слов, то указатель кучи перемещается на N слов, а его прежнее значение возвращается в качестве указателя на вновь размещенный элемент. Когда освобождается память позади стоящего впереди указателя кучи, она может быть объединена в список свободного пространства.

Рано или поздно указатель кучи достигнет конца блока, составляющего кучу. Теперь следует повторно использовать ту часть памяти, которая на данный момент не задействована под активные элементы. С учетом того, что эти элементы имеют различный размер, можно предложить два способа повторного использования этой памяти:

1) использовать список свободного пространства, проводя в нем поиск блока подходящего размера и возвращая после выделения памяти оставшуюся часть блока снова в список свободного пространства;
2) уплотнить свободное пространство, переместив все активные элементы к одному краю кучи, оставляя при этом все свободное пространство в виде единого блока и переустанавливая указатель кучи на начало этого блока.

Рассмотрим каждый из этих способов по очереди.

Повторное использование свободного пространства

При получении запроса на элемент размером N слов самым простым способом его выполнения является поиск в списке свободного пространства блока, состоящего из N и более слов. Блок из N слов можно прямо выделить для удовлетворения запроса. Блок, состоящий из более чем N слов, следует разбить на два блока: один, состоящий из N слов, который тут же выделяется для удовлетворения запроса, и оставшийся блок, который возвращается в список свободного пространства. Используется несколько различных методов непосредственного распределения памяти из такого списка свободного пространства.

1. *Метод первого подходящего.* Если требуется блок размером в N слов, список свободного пространства просматривается до обнаружения *первого* блока, состоящего из N или более слов, который затем разбивается на блок

из N слов и остаток, который возвращается в список свободного пространства.

2. *Метод наиболее подходящего*. Если требуется блок размером в N слов, список свободного пространства просматривается до обнаружения *минимального* блока, количество слов в котором больше или равно N. Если в этом блоке в точности N слов, то он распределяется целиком как единое целое; если же количество слов в нем больше N, то он разбивается на две части и остаток возвращается в список свободного пространства.

Сохранение элементов в списке свободного пространства в порядке возрастания их размеров делает распределение достаточно эффективным. В этом случае надо сканировать список, пока не будет найден блок требуемого размера. Однако возрастает стоимость добавления элемента в список свободного пространства, так как приходится осуществлять поиск в списке подходящего места для добавления нового элемента.

Восстановление блоков памяти переменного размера

Перед тем как рассматривать задачу уплотнения памяти, рассмотрим методы восстановления памяти в том случае, когда куча состоит из блоков переменного размера. Отличия от случая блоков постоянного размера невелики. Явный возврат освобождаемой памяти в список свободного пространства является простейшим методом, но по-прежнему присутствуют проблемы накопления мусора и появления повисших ссылок. Счетчики ссылок в этом случае могут использоваться обычным образом.

Сбор мусора также представляет подходящий метод, хотя в случае блоков переменного размера возникает несколько дополнительных проблем. Как и раньше, процедура сбора мусора состоит из двух стадий — стадии маркировки и следующей за ней стадии сбора. Маркировка должна основываться на той же методике следования по цепи указателей. Сложности появляются на стадии сбора. Раньше эта задача решалась путем простого последовательного просмотра памяти и проверки бита сбора мусора каждого конкретного элемента. Если этот бит находился в состоянии «включен», элемент возвращался в список свободного пространства; если бит был «выключен», то это означало, что элемент в данный момент активен и его нужно пропустить. В случае элементов переменного размера хотелось бы использовать эту же схему, но теперь у нас встает проблема определения границы между элементами. Где заканчивается один элемент и начинается другой? Без этой информации невозможно выполнить сбор мусора.

Самым простым способом решения этой проблемы является хранение в первом слове каждого блока (и активного, и неактивного), наряду с битом сбора мусора, целочисленного *идентификатора длины* этого блока, который просто указывает, сколько слов содержится в данном блоке. При наличии идентификатора длины мы вновь можем последовательно просматривать блоки памяти, анализируя только первое слово каждого блока. Во время этого просмотра, прежде чем возвращать освободившиеся свободные блоки в список свободного пространства, смежные свободные блоки можно объединять в один блок, тем самым устраняя проблему частичного уплотнения, обсуждаемую ниже.

Сбор мусора можно также эффективно объединить с полным уплотнением, чтобы вовсе исключить необходимость в списке свободного пространства. В этом случае будет нужен только простой указатель кучи.

Уплотнение и проблема фрагментации памяти

Проблема, с которой мы сталкиваемся при использовании элементов переменного размера в любой системе управления кучей, — это проблема *фрагментации*. Мы начинаем с одного большого блока свободного пространства. В ходе вычислений этот блок постепенно разбивается на меньшие блоки посредством операций выделения памяти, ее восстановления и повторного использования. Если для распределения памяти применяется только простой метод первого подходящего или наиболее подходящего блока, то очевидно, что блоки свободного пространства продолжают расщепляться на все более мелкие куски. В конце концов наступит момент, когда программа распределения памяти не сможет удовлетворить запрос на блок из N слов, поскольку не окажется ни одного достаточно большого блока, хотя в целом список свободного пространства может содержать гораздо более чем N слов свободной памяти. Без некоторого объединения (уплотнения) свободных блоков в блоки большего размера выполнение программы будет остановлено из-за нехватки свободной памяти раньше, чем реально закончатся ресурсы памяти.

В зависимости от того, можно ли перемещать активные блоки в куче, применяется один из двух возможных подходов к уплотнению.

1. *Частичное уплотнение*. Если активные блоки *нельзя* перемещать (или перемещение обходится слишком дорого), то можно объединять только смежные блоки из списка свободного пространства.
2. *Полное уплотнение*. Если активные блоки *можно* перемещать, тогда все активные блоки могут быть перемещены в один конец кучи, при этом все свободное пространство оказывается сосредоточенным на другом конце в одном непрерывном блоке. Полное уплотнение подразумевает, что при перемещении активного блока все указатели на него модифицируются таким образом, чтобы указывать на его новое местоположение.

10.5. Рекомендуемая литература

В большей части текстов, посвященных конструированию компиляторов, рассматривается стратегия статического и стекового распределения памяти (см. ссылки в главе 3). Методы управления кучей изучались достаточно широко — см., например, [101]. В [29] представлен краткий обзор методов сбора мусора. В [85] описан метод выполнения анализа процесса компиляции, который позволяет избежать выполнения процедуры сбора мусора во время выполнения.

В операционных системах часто предусмотрены некоторые возможности управления памятью, которые пересекаются с аналогичными возможностями языков программирования. В настоящее время во многие операционные системы встроены *системы виртуальной памяти*, которые распределяют память в виде *страниц* фиксированного размера или *сегментов* переменного размера. Эти возможности иногда оказываются полезными, но обычно они не могут полностью заменить уп-

равление кучей, которое предусмотрено в реализации языка программирования. Управление памятью при помощи операционной системы — тема многих работ по операционным системам.

10.6. Задачи и упражнения

1. Проанализируйте методы управления памятью, применяемые в доступных вам реализациях языков. Рассмотрите различные элементы, которые должны быть представлены в памяти во время выполнения программы (перечисленные в разделе 10.1). Имеются ли какие-либо другие крупные структуры, которые требуют представления в памяти, помимо перечисленных в разделе 10.1?
2. Существует альтернативный способ сбора мусора, в котором используются две области памяти. Во время первой фазы сбора мусора, *маркировки*, каждый объект, на который имеется ссылка, копируется в голову второй области. Таким образом, как только все активные узлы помечены, сбор мусора можно считать выполненным. Теперь распределение памяти осуществляется из второй области. Когда эта область заполняется, фаза маркировки копирует активные элементы снова в первую область. Сравните этот алгоритм с двухпроходным алгоритмом (маркировка-сбор), описанным в разделе 10.4.2.
3. Проанализируйте элементарные операции языка, который вам хорошо знаком. Какие операции требуют выделения или освобождения памяти? Всегда ли используются блоки памяти одного размера или иногда требуются блоки переменных размеров? Когда происходит выделение и освобождение памяти — только при входе и выходе из подпрограмм или в произвольных точках во время выполнения? Что можно сказать о структурах управления, используемых в данной реализации языка, на основании общей модели выделения и освобождения памяти? Можете ли вы доказать, что требуется куча с переменным или, напротив, с фиксированным размером элементов? Можно ли доказать, что достаточно только центрального стека?
4. Как показано на рис. 9.3, *б*, в большинстве реализаций языка Pascal используется единый блок памяти как для центрального стека, так и для кучи. Во время выполнения стек и куча растут навстречу друг другу с противоположных концов блока памяти, выделяя память из свободного пространства, расположенного между ними. Предположим, что стек и куча встречаются друг с другом, так как свободное пространство полностью использовано. Объясните доступные для разработчика транслятора варианты решения, если:

   а) следующий запрос — это запрос на новый блок памяти из стека;

   б) следующий запрос — запрос на новый блок памяти из кучи.
5. Одно из удивительных свойств сбора мусора как метода восстановления памяти — это то, что его стоимость *обратно пропорциональна* количеству восстановленной памяти (то есть чем меньше памяти восстанавливается, тем дороже обходится сбор мусора).

а) Объясните, в чем причина такой обратной пропорциональности.

б) Обычно, когда ресурсы памяти почти исчерпаны, программа выполняет несколько раз дорогостоящую и занимающую много времени процедуру сбора мусора (которая восстанавливает малый объем памяти), прежде чем окончательно остановиться. Предложите метод, позволяющий избежать многократного выполнения таких бесполезных процедур сбора мусора.

6. В одном из ранних расширений языка FORTRAN, предназначенном для обработки списков, у каждого списка имеется специальная голова, в которой содержится счетчик ссылок. Когда какой-нибудь список освобождается, то вместо возвращения его элементов в голову списка свободного пространства возвращается только его головной элемент, причем он помещается в конец списка свободного пространства (для этого используется специальный указатель на конец списка свободного пространства). Таким образом минимизируется стоимость возвращения списка в свободное пространство. Элементы списка возвращаются в список свободного пространства только тогда, когда голова списка достигнет вершины списка свободного пространства. Какие преимущества дает такой метод, осуществляя перенос затрат на освобождение элементов списка с момента *восстановления* списка на момент *повторного использования* памяти?

7. Элементы данных размером в *целое слово*, например числа, представляют проблемы при сборе мусора. Обычно такие элементы данных занимают целиком элемент кучи, не оставляя места для дополнительных битов маркировки. В таких случаях все подобные элементы размером в одно слово располагаются в специальном разделе кучи, а все биты сбора мусора для этих элементов записываются в специальный упакованный массив (*битовый вектор*), расположенный за пределами кучи. Предполагая, что все элементы размером в целое слово являются числами (и, следовательно, не содержат указателей на другие элементы кучи), разработайте алгоритм сбора мусора, в котором допускаются указатели на элементы кучи размером в целое слово. Алгоритм должен включать маркировку и сбор. Организуйте поддержку отдельного списка свободного пространства для той части кучи, которая содержит элементы размером в целое слово.

8. В тексте утверждается, что в приложениях параллельной обработки предпочтительнее использовать счетчики ссылок для сбора мусора динамически выделяемой памяти. Почему в этой среде использование алгоритма маркировка-сбор было бы менее эффективным?

9. Предложите алгоритм для *накопления* блоков, помеченных во время сбора мусора в куче с элементами переменного размера. Предполагается, что первое слово каждого блока содержит бит сбора мусора и идентификатор длины. Во время сбора мусора следует объединять (уплотнить) все смежные свободные блоки.

10. Разработайте систему управления распределением памяти для кучи В, состоящей из блоков переменного размера, при выполнении следующих предположений.

а) Куча B используется только для хранения массивов вещественных чисел (одно число располагается в одном слове). Каждый массив содержит как минимум два элемента.

б) Доступ к любому блоку массива осуществляется только через единственный внешний указатель, хранящийся в дескрипторе массива. Все дескрипторы массивов хранятся в блоке A, расположенном отдельно от кучи B.

в) Во время выполнения запросы на выделение блоков из кучи происходят случайным образом (и достаточно часто).

г) Блоки явным образом возвращаются в свободное состояние при разрушении массивов. Это также происходит в случайные моменты времени и достаточно часто.

д) Постоянные потери памяти вследствие ее фрагментации недопустимы. (Заметим, что свободный блок из одного слова никогда не может быть использован повторно.)

Эти предположения в основном реализованы в APL. Ваш проект должен специфицировать:

1) *начальную организацию* блока кучи B, а также все специальные внешние структуры, необходимые для управления памятью (например, голову списка свободного пространства);
2) *механизм выделения памяти* на запрос блока длиной N слов из кучи B;
3) *механизм восстановления памяти* при обращении к менеджеру памяти для освобождения блока из M слов с заданным адресом;
4) *механизм уплотнения памяти* (если он необходим), включая спецификацию его работы и моментов времени, когда он вызывается.

11. В различных реализациях языка Pascal операция `dispose` обрабатывается различными способами. Ниже перечислены применяемые методики обработки:

а) Операция `dispose(P)` всего лишь присваивает значение `nil` переменной-указателю `P`. Предложите реализацию управления кучей с использованием одного указателя на текущую вершину (первое свободное место) кучи. Перемещайте указатель на новое место для выделения памяти, когда выполняется операция `new`. Не допускается перемещать указатель в обратном направлении.

б) Предложите реализацию управления кучей с использованием, как в предыдущем случае, единственного указателя, который передвигается вперед в результате выполнения операции `new`, однако теперь операция `dispose` не только присваивает значение `nil` указателю `P`, но и перемещает указатель вершины кучи назад на количество элементов, занимаемых объектом данных, на который указывает `P`, так что эта область памяти восстанавливается и возвращается в кучу. (*Подсказка*: рассмотрите последовательность `new(P); new(Q); dispose(P);`.)

в) Предложите способ реализации кучи с использованием списка свободного пространства для блоков памяти переменного размера, как описано

в разделе 10.4. Используйте стратегию первого подходящего или наиболее подходящего блока для выделения памяти, запрошенной операцией new. Реализуйте операцию dispose так, чтобы она устанавливала свой параметр-указатель равным значению nil и возвращала блок памяти, на который указывает параметр-указатель P, в список свободного пространства.

Рассмотрите проблемы мусора, повисших ссылок и фрагментации для каждого из приведенных способов реализации операций dispose и new.

12. Для создания связанных списков удобно применять указатель this. Рассмотрим следующее определение класса (class):

```
class LinkedList
{public: AddToList(LinkedList *X) // Добавляем X к списку
private: LinkedList *PreviousListItem, *NextListItem;};
```

Разработайте код для метода AddToList, который вставляет в список переданный ему аргумент в позицию, следующую за текущей (отмеченной указателем this), как это делается в коде:

```
LinkedList A[10];
int i, j;
...
//Вставляем A[j]после A[i]
A[i].AddToList(&A[j])
```

Глава 11. Распределенная обработка данных

Технология компьютерных вычислений прошла путь от замкнутой работы на одной машине в 50-х и 60-х гг. до интеграции совместных работ, часто проводимых в разных точках земного шара. Системы клиент-сервер, локальные сети и Всемирная паутина WWW — это примеры того, когда множество компьютеров, находящихся в разных местах, должно взаимодействовать между собой для того, чтобы обработать определенную информацию. Основная проблема, которую пытаются решить перечисленные технологии, заключается в том, что отдельный компьютер не является достаточно большим и быстрым, чтобы решить стоящие перед нами современные вычислительные задачи, что неизбежно приводит к совместному использованию нескольких компьютеров для решения одной задачи. В этой главе мы рассмотрим эволюцию идеологии языков программирования, позволившую осуществить переход к распределенной обработке данных.

11.1. Различные варианты управления подпрограммами

В этом разделе мы обсудим различные варианты конструирования подпрограмм. Активация процедур по методу простого вызова-возврата (см. раздел 9.1) со статическими и динамическими связями с объектами данных основывалась на правилах области видимости, что позволило нам реализовать многие языки программирования. Но для того, чтобы наша структура подпрограммы предоставляла некоторые дополнительные возможности, например реализацию сопрограмм, исключений и задач, нам следует рассмотреть некоторые вариации этого основного механизма.

В первую очередь вспомним, что связь подпрограмм, описанная в разделе 9.1, основывалась на следующих пяти предположениях:

1) подпрограммы не могут быть рекурсивными;
2) требуется явный оператор вызова подпрограммы;
3) подпрограммы должны полностью выполняться при каждом вызове;
4) управление в подпрограмму должно передаваться непосредственно в точке ее вызова;
5) должна существовать одна последовательность выполнения.

Первое предположение, касающееся рекурсии, мы уже рассмотрели достаточно подробно. Теперь мы попробуем изменить другие предположения и исследуем форму механизма подпрограмм, к которым приводят эти изменения.

11.1.1. Исключения и обработчики исключений

Во время выполнения программы часто встречаются события и условия, которые могут рассматриваться как исключения. Вместо нормального продолжения выполнения программы в этом случае необходимо вызвать подпрограмму, которая осуществляет специальную обработку, включая способность справляться со следующими ситуациями.

1. *Сбойные ситуации*, связанные с возникновением таких ошибок, как, например, переполнение при выполнении арифметической операции или ссылка на элемент массива с индексом вне допустимого диапазона.
2. *Непредсказуемые ситуации*, возникающие при нормальном выполнении программы, как, например, генерация специальных заголовков в конце выводимой на принтер страницы или обработка признака конца вводимого файла.
3. *Трассировка и мониторинг* во время тестирования программы, например вывод трассировочной печати, когда меняется значение какой-либо переменной.

Хотя в программу обычно можно вставить явный тест отслеживания подобных исключительных ситуаций, такие дополнительные операторы могут делать неясной основную структуру программы. Проще смягчить требование, гласящее, что подпрограммы должны вызываться исключительно с помощью явного оператора вызова и обеспечить возможность вызова подпрограммы в случае выполнения определенного условия или возникновения определенного события. Это условие или событие обычно называются *исключением*, или *сигналом*, а подпрограмма, выполняющая их специальную обработку, — *обработчиком исключений*. Действия, связанные с отслеживанием появления исключения, прерыванием работы программы и передачей контроля обработчику исключений, называются *генерацией исключения*. В языке Ada существует класс исключений, называемых *проверками*, или условиями, которые требуют выполнения определенного кода. Например, `Index_Check` — это исключение `Constraint_Error`, которое генерируется, когда программа выходит за границы массива.

Обработчики ошибок

Поскольку обработчик ошибок вызывается без всякого явного оператора, обычно ему не требуется ни имя, ни параметры. Определение обработчика ошибок обычно включает в себя только лишь

1) набор объявлений локальных переменных (если необходимо);
2) последовательность выполняемых операторов.

Для того чтобы установить связь между исключениями и их обработчиками, каждому классу исключений присваивается некоторое имя. Некоторые исключения в языке могут быть определенны заранее (например, `Constraint_Error`, `Program_Error`,

Storage_Error или Numeric_Error в Ada). Другие могут быть определены программистом (например, в программу может быть включено объявление Underflow: exception или Overflow: exception). Каждый обработчик исключений затем сопоставляется с именем (или именами) тех исключений, которые он должен обрабатывать. Обычно все обработчики исключений группируются в начале или в конце большой программы или подпрограммы, где может произойти исключительная ситуация. Типичной в этом отношении является структура языка Ada:

```
procedure Sub is
   Bad_Data_Value: exception:
         -другие объявления в Sub
begin
         -операторы. выполняемые при нормальном ходе Sub
exception
   when Bad_Data_Value =>
      -обработчик для некорректных значений данных
   when Constraint_Error =>
      -обработчик для предопределенного исключения Constraint_Error
   when others =>
      -обработчик для всех других исключений
end;
```

Генерация исключения. Исключительная ситуация может быть сгенерирована какой-либо элементарной операцией, определенной в языке (например, операция сложения или умножения может сгенерировать исключение Constraint_Error). С другой стороны, исключение может быть сгенерировано явным образом программистом с помощью оператора, специально предназначенного для этой цели, как, например, следующий оператор Ada:

```
raise Bad_Data_Value;
```

который может быть выполнен в подпрограмме, если окажется, что какая-либо переменная или входной файл содержат некорректное значение.

Если в подпрограмме используется явный оператор генерации исключения raise и в ней имеется обработчик этого исключения, как, например, в случае, когда оператор

```
if X = 0 then raise Bad_Data_Value end if;
```

появляется в теле приведенной выше процедуры Sub, то оператор raise передает управление соответствующему обработчику исключений, который затем осуществляет выход из процедуры. В языках, в которых отсутствуют возможности явной генерации исключений, для передачи управления коду, обрабатывающему исключение, используется оператор goto, как отмечалось в разделе 8.2.

Обработчику исключений можно передавать данные, как показано в следующем примере из языка ML. Нормальным действием в ML является завершение процедуры при возникновении исключительной ситуации. Если же в ней присутствует обработчик исключений, то после его выполнения управление возвращается в подпрограмму в ту точку, где была обнаружена ошибка, как, например, в следующем случае:

```
exception BadDenominator of int;
fun InnerDivide(a:int,b:int):real=
      if b=0 then raise BadDenominator(b)
         else real(a)/real(b);
fun Divide(a,b)=InnerDivide(a,b) handle
BadDenominator(b)=>(print(b); "попытка деления на 0"; 0.0);
```

В данном случае программа вызывает функцию Divide, которая, в свою очередь, вызывает функцию InnerDivide. В случае, если знаменатель равен нулю, функция InnerDivide обращается к обработчику исключений BadDenominator, который печатает сообщение об ошибке, возвращает вещественное значение 0.0 функции Divide, и выполнение продолжается. (Отметим, что для функции Divide не требуется указывать типы аргументов и возвращаемого результата. ML пытается сделать предположение о корректном типе, и в данном случае он может сделать это, используя явное определение типов, данное для функции InnerDivide.)

В C++ (хотя не в каждом трансляторе C++ реализованы исключения) и Java исключения обрабатываются при помощи оператора try. C++ генерирует исключение, порождая[1] (throwing) его, а обрабатывает исключение посредством его перехвата (catching). Синтаксис для вызова и обработки исключений в C++ похож на синтаксис ML, за исключением того, что после обработки исключения выполнение программы завершается:

```
try {
      statement1:
      statement2:
      ...
      if BadCondition {throw ExceptionName}:
         }
catch ExceptionName {                  // Действия по обработке исключения
         }                             // Конец исключения
      }
```

Распространение исключения. Часто при проектировании программы выясняется, что место возникновения исключения не является лучшим местом для его обработки. Например, назначением некоторой подпрограммы может быть считывание значений данных из входного файла и передача их группе вложенных подпрограмм для дальнейшей обработки. Предположим, что во входном файле могут содержаться неверные значения данных нескольких различных типов, и каждая из подпрограмм проверяет свой тип ошибок в данных входного файла, но результат всех этих проверок один и тот же: печатается сообщение об ошибке и файл продолжает обрабатываться дальше, минуя неправильные значения. В таком случае было бы правильнее, если бы обработчик ошибок являлся частью подпрограммы, которая считывает файл, а каждая из подпрограмм могла бы порождать исключение Bad_Data_Value. Когда исключение обрабатывается в подпрограмме, отличной от той, которая его породила, то говорят, что исключение *распространяется* от точки его порождения до точки его обработки.

Правило, определяющее, какой обработчик должен обрабатывать конкретное исключение, обычно определяется в терминах *динамической цепи* активаций подпрограмм, которая ведет к подпрограмме, породившей исключение. Если исключение P порождено в подпрограмме C, то оно обрабатывается обработчиком, определенным в этой подпрограмме C (если таковой имеется). Если обработчика нет,

---

[1] В C++ и Java исключение представляется специальным объектом, который существует, пока его не перехватит какой-нибудь обработчик. Отсюда в англоязычной технической литературе используется термин «throw» («выбросить») для обозначения порождения исключения – процедура «выбрасывает» объект-исключение, который должен быть «пойман» каким-либо обработчиком, не обязательно определенным в этой же процедуре. — *Примеч. науч. ред.*

то C завершает свое выполнение. Если C была вызвана из подпрограммы B, то исключение распространяется на B и снова порождается в той точке подпрограммы B, откуда вызывалась подпрограмма C. Если в B нет обработчика для исключения P, то B также завершается, а исключение распространяется на подпрограмму, вызвавшую B и т. д. Если ни в одной подпрограмме, а также в главной программе не найдется обработчика для этого исключения, то вся программа останавливается и вызывается стандартный обработчик, определенный в данном языке программирования. В частности, в предыдущем примере из ML функция InnerDivide породила исключение BadDenominator, которое было распространено до обработчика исключения в функции Divide, который затем вернул определенное программистом значение как результат обработки этого исключения.

Важной особенностью этого правила распространения исключений является то, что оно позволяет подпрограмме оставаться определенной пользователем абстрактной операцией даже в случае обработки исключений. Элементарная операция или подпрограмма могут внезапно прервать свое нормальное выполнение и породить исключение. Для вызывающей подпрограммы эффект порождения исключения подпрограммой и элементарной операцией будет одинаковым, если подпрограмма сама не обрабатывает это исключение. Если же это исключение обрабатывается внутри подпрограммы, то она завершается нормальным образом, и вызывающая подпрограмма даже не заметит, что имела место исключительная ситуация.

После обработки исключения. После того как обработчик завершил обработку исключения, возникает несколько неприятный вопрос: куда должно быть передано управление, так как для обработчиков не существует явного вызова? Должно ли управление передаваться в точку порождения исключения (эта точка может отстоять от точки обработки на несколько уровней вызовов подпрограмм)? Или управление следует передать оператору содержащей обработчик исключения подпрограммы, в котором исключение было порождено снова после его распространения? Должна ли подпрограмма, содержащая обработчик, завершиться, но завершиться нормально, так что вызвавшей ее подпрограмме будет «казаться», что ничего и не произошло? В языке Ada, например, именно последнее решение и используется; в ML предусмотрено несколько различных вариантов, а в других языках предлагаются иные решения.

Реализация

Исключительные ситуации могут иметь два различных источника:

1) условия, обнаруженные виртуальной машиной;

2) условия, сгенерированные семантикой языка программирования.

В первом случае исключения операционной системы могут быть порождены непосредственно аппаратными прерываниями, например арифметическим переполнением, либо они могут быть обусловлены вспомогательным программным обеспечением, например условием достижения конца файла. В C программист имеет прямой доступ к этим сигналам, обрабатываемым операционной системой. Программист может *разрешить* прерывание (например, при помощи функции sigaction в UNIX, которая определяет процедуру, вызываемую при возбуждении определенного сигнала).

В языке программирования могут быть предусмотрены дополнительные исключения — для этого транслятор языка вставляет дополнительные инструкции в выполняемый код. Например, чтобы обнаружить исключение Index_Check, причиной которого явился выход индекса элемента массива за допустимый диапазон его изменения, транслятор вставляет явную последовательность команд при каждой ссылке на массив (например, A[I, J]), которая определяет, находятся ли значения I и J в объявленных границах массива. Таким образом, если только проверка исключений не обеспечивается аппаратной частью компьютера или операционной системой, проверка исключений требует определенного программного моделирования. Часто затраты на такую проверку достаточно велики — как в отношении времени выполнения, так и в отношении объема памяти, необходимой для хранения кода. Например, может оказаться, что проверка того, попадают ли индексы для элемента массива A[I, J] в установленный диапазон, займет больше времени, чем сама операция извлечения этого элемента. По причине этих дополнительных затрат большинство языков предусматривают средство отключения проверки возникновения исключений в тех частях программы, где, по мнению программиста, это абсолютно безопасно (например, конструкция pragma Surpress(Index_Check) в языке Ada).

Если в программе сгенерировано исключение, передача управления обработчику исключений, содержащемуся в этой же программе, обычно реализуется командой перехода непосредственно на начало кода обработчика. Распространение исключений от точек порождения вперед по динамической цепи вызовов подпрограмм можно реализовать, используя динамическую цепь, сформированную точками возврата в записях активаций подпрограмм из центрального стека, как обсуждалось ранее. По мере распространения исключения по динамической цепи каждая активация подпрограммы должна быть завершена при помощи специальной формы команды *возврата*, которая возвращает управление вызывающей подпрограмме и снова порождает исключение, но уже в вызывающей подпрограмме. Эта последовательность возвратов продолжается до тех пор, пока динамическая цепочка вызовов подпрограмм не приведет нас к той подпрограмме, в которой присутствует обработчик сгенерированного исключения.

Когда нужный обработчик найден, он вызывается точно так же, как вызывается обычная подпрограмма. Когда обработчик завершает свое выполнение, он, однако, может также завершить содержащую его подпрограмму, приводя, таким образом, к двум нормальным возвратам из подпрограмм, следующим один немедленно за другим. Когда таким способом будет «пройдена» динамическая цепочка до последнего нормального возврата в подпрограмму, то именно эта подпрограмма продолжает свое дальнейшее выполнение обычным образом.

Утверждения

Понятие утверждения (assertion) является родственным понятию исключения. Утверждение — это просто некоторый оператор, предполагающий некоторые отношения между объектами данных в программе, как, например, следующий оператор C++:

```
#include < assert.h >
...
assert (x>y+1);
```

Предопределенный макрос[1] С++ генерирует следующий условный оператор:

```
if (x>y+1) {                    /* Печать сообщения об ошибке */ }
```

Утверждение — это удобный способ тестирования программы на наличие ошибок без сложного кодирования, усложняющего исходный текст программы. Когда разработка программы завершена, утверждение assert может остаться в программе в качестве документации (см. раздел 4.2.4), а макрос можно изменить таким образом, чтобы он не генерировал никаких операторов и оставался в программе просто в виде комментария.

11.1.2. Сопрограммы

Допустим, что мы отказались от ограничения 3 из раздела 11.1, то есть разрешили передавать управление из вызванной подпрограммы в вызывающую, до того как вызванная подпрограмма завершит свое выполнение нормальным образом. Такие подпрограммы называются *сопрограммами*. Кода сопрограмма получает управление от другой подпрограммы, она выполняется лишь частично, а затем приостанавливается и возвращает управление вызвавшей ее подпрограмме. Впоследствии в некоторой точке вызывающая программа может *возобновить* (resume) выполнение сопрограммы с той точки, в которой выполнение было ранее приостановлено.

Отметим симметрию, введенную здесь в структуру вызывающей и вызываемой подпрограмм. Если A вызывает подпрограмму B как сопрограмму, то B выполняется некоторое время и возвращает управление в подпрограмму A, как сделала бы любая обычная подпрограмма. Когда A снова передает управление B при помощи оператора resume B, то B снова выполняется некоторое время и возвращает управление в подпрограмму A, как обычная подпрограмма. Таким образом, для A сопрограмма B выглядит как обычная подпрограмма. Но если посмотреть на этот процесс с другой точки зрения, из подпрограммы B, то мы увидим аналогичную ситуацию. Подпрограмма B в середине своего выполнения *возобновляет* выполнение A. Подпрограмма A выполняется некоторое время и возвращает управление B. Подпрограмма B продолжает свое выполнение в течение некоторого времени и возвращает управление A. Подпрограмма A продолжает свое выполнение в течение некоторого времени и возвращает управление B. A выглядит с точки зрения подпрограммы B как вполне обычная подпрограмма. Название *сопрограмма* происходит именно от этой описанной симметрии. Взаимоотношение между двумя сопрограммами строится не по принципу родитель–потомок или вызывающий–вызываемый; это скорее взаимоотношение двух равных структур, которые передают друг другу управление по мере выполнения, причем ни одна из них не контролирует другую явным образом (рис. 11.1).

В настоящее время сопрограммы являются общей структурой управления только в языках дискретного моделирования (см. раздел 11.1.3). Тем не менее для многих алгоритмов они являются более естественной структурой управления, чем обычная иерархия вызовов подпрограмм. Более того, простая структура сопрограмм во многих языках легко может быть смоделирована с помощью оператора goto и пе-

[1] В С++ assert является макроопределением. — *Примеч. науч. ред.*

ременной *точки возобновления*, определяющей метку того оператора, с которого должно возобновиться выполнение.

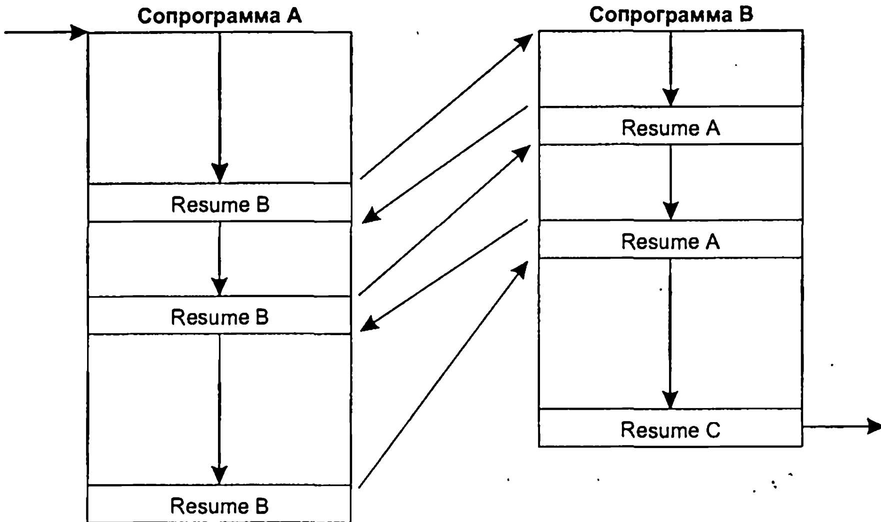


Рис. 11.1. Передача управления между сопрограммами

Реализация. Команда resume, которая передает управление от одной сопрограммы другой, предписывает возобновление некоторой определенной активации сопрограммы. Если существует несколько рекурсивных активаций сопрограммы B, то оператор resume B не имеет точного значения. По этой причине проще всего представлять себе сопрограммы в контексте, когда в каждый момент времени существует только одна активация данной сопрограммы. Это ограничение позволяет нам использовать для реализации сопрограмм технику, аналогичную той, которая использовалась для простых структур вызова-возврата (см. раздел 8.3.2). В начале выполнения одна запись активации статически размещается в памяти в виде расширения сегмента кода сопрограммы. В записи активации резервируется специальное поле, называемое теперь *точкой возобновления*, для сохранения старого значения ip указателя CIP, когда команда resume передает управление другой сопрограмме. Но в отличие от точки возврата простой подпрограммы, это поле точки возобновления в программе B используется для хранения значения ip *самой подпрограммы* B. Выполнение оператора resume B в сопрограмме A происходит в два этапа.

1. Текущее значение CIP сохраняется в поле точки возобновления записи активации сопрограммы A.
2. Значение ip, хранящееся в поле точки возобновления сопрограммы B, извлекается из записи активации сопрограммы B и присваивается указателю CIP, чтобы инициировать передачу управления правильной команде сопрограммы B.

Так как не существует явной команды возврата, то сопрограмме B нет необходимости «знать», что сопрограмма A передала ей управление.

Структуры управления, в которых подпрограммы могут вызываться и как сопрограммы, и как обычные подпрограммы и в которых сопрограммы могут быть рекурсивными (то есть иметь несколько одновременно существующих активаций), требуют более сложной реализации.

11.1.3. Планируемые подпрограммы

Концепция *планирования подпрограмм* основывается на отказе от предположения, что подпрограмма всегда должна начинать свое выполнение непосредственно в точке вызова. Оператор вызова обычной подпрограммы можно рассматривать как оператор, указывающий, что вызываемая подпрограмма должна немедленно начать выполнение, без ожидания завершения выполнения вызывающей программы. При этом завершение выполнения вызывающей программы перепланируется на момент завершения вызванной подпрограммы. Структура управления обработки исключений также может рассматриваться как средство планирования подпрограмм. Обработчик исключения выполняется всякий раз, когда порождается определенное исключение.

Продолжая делать обобщения, приходим к следующим возможным способам планирования подпрограмм.

1. Планировать выполнение подпрограмм следует таким образом, чтобы одна подпрограмма выполнялась до или после выполнения других подпрограмм, например, оператор `call B after A` мог бы означать, что выполнение подпрограммы B запланировано после завершения выполнения подпрограммы A.

2. Планировать выполнение программ в зависимости от истинности какого-либо булева выражения, например:

   ```
   call B when X = 5 and Z > 0
   ```

   Такое планирование является одним из видов обобщенной обработки исключений; подпрограмма B вызывается всякий раз, когда значения переменных Z и X изменяются и удовлетворяют заданным условиям.

3. Планировать подпрограммы на основе моделирования *временной шкалы*, например:

   ```
   call B at time = 25 or call B at time = CurrentTime + 10
   ```

   Это позволяет чередовать запланированные вызовы подпрограмм из разных источников.

4. Планировать подпрограммы в соответствии с назначенным приоритетом, например, следующий оператор:

   ```
   call B with priority 7
   ```

   активизировал бы подпрограмму B, если не запланирован вызов ни одной подпрограммы с более высоким приоритетом.

Реализованные способы обобщенного планирования подпрограмм можно встретить только в языках моделирования дискретных систем, таких как GPSS, SIMSCRIPT и SIMULA, хотя перечисленные выше концепции имеют широкое применение. Каждый из описанных способов планирования подпрограмм встречается по мень-

шей мере в одном из перечисленных языков. Наиболее важным методом, используемым при моделировании систем, является третий — планирование, основанное на моделировании временной шкалы. Этот способ планирования мы и рассмотрим несколько подробнее.

Когда мы говорим о планировании подпрограмм, мы имеем в виду планирование *активаций* подпрограмм, поскольку это планирование относится ко времени выполнения программы, для которого может быть спланировано множество активаций одной и той же подпрограммы в разных точках. Планирование подпрограмм в наиболее общем смысле означает, что главная программа уже не пишется программистом, а является системной *программой-планировщиком*, и ее назначение заключается в том, чтобы поддерживать список текущих запланированных подпрограмм, упорядоченный в соответствии с последовательностью их выполнения. В языке предусматриваются операторы, которые позволяют вставлять активации подпрограмм в этот список во время выполнения. Программа-планировщик вызывает подпрограммы в указанной последовательности. Когда заканчивается выполнение одной подпрограммы, инициируется выполнение следующей подпрограммы из этого списка. Обычно допускается и обычный вызов подпрограмм, иногда просто путем временной приостановки выполнения одной подпрограммы и немедленного выполнения другой.

В языках моделирования наиболее общий подход к планированию подпрограмм основывается на обобщенной концепции сопрограммы. Выполнение активации одной подпрограммы состоит из набора *активных* и *пассивных* фаз. Во время активной фазы подпрограмма имеет управление и выполняется; во время пассивной фазы подпрограмма передает управление куда-то в другое место и ожидает возобновления своего выполнения. Но в данном случае сопрограмма передает управление другой сопрограмме не напрямую в момент перехода от активной к пассивной фазе — сначала управление возвращается программе-планировщику, которая затем уже передает управление следующей подпрограмме из списка. Эта передача управления может принимать форму возобновления подпрограммы, если она уже была вызвана, а затем приостановлена, или инициирования совершенно новой активации подпрограммы.

Концепция планирования сопрограмм представляется чрезвычайно простой в случае моделирования временной шкалы. Предположим, что каждая активная фаза выполнения подпрограммы может быть спланирована таким образом, чтобы она начиналась в какой-либо точке целочисленной шкалы времени, начинающейся в момент времени `T = 0`. `T` — это целочисленная переменная, которая всегда содержит значение текущего времени на модели шкалы времени. Выполнение активной фазы подпрограммы на этой шкале всегда происходит мгновенно (то есть значение переменной `T` не меняется во время выполнения активной фазы подпрограммы). Когда активная фаза завершается и управление переходит к программе-планировщику, планировщик обновляет значение `T`, присваивая этой переменной значение времени, соответствующее времени вызова следующей подпрограммы из списка планируемых подпрограмм, и передает управление этой подпрограмме. Вновь вызванная подпрограмма частично выполняется и возвращает управление планировщику, который снова обновляет значение `T` и активизирует следующую подпрограмму из списка.

11.2. Параллельное программирование

Последнее ограничение, указанное в разделе 9.1, которое нам предстоит рассмотреть, заключается в том, что при выполнении программ существует единственная последовательность выполнения. Вообще, несколько подпрограмм могут выполняться *одновременно*. Когда существует единственная последовательность, программа называется *последовательной программой*, поскольку выполнение ее подпрограмм происходит в заранее определенной последовательности. В более общем случае программа называется *параллельной программой*. Каждая подпрограмма, которая может выполняться параллельно с другими подпрограммами, называется *задачей* (или иногда *процессом*).

До сих пор при обсуждении управления последовательностью действий мы предполагали, что при выполнении программы всегда существует некоторый предопределенный поток управления, описывающий выполнение программы. Это хорошо согласуется с неймановской архитектурой, которую мы предполагали для нашего реального компьютера. Тем не менее в программировании параллельные вычисления также играют важную роль.

В настоящее время достаточно широко используются компьютерные системы, допускающие параллельное программирование. В *мультипроцессорной системе* имеется несколько центральных процессоров, которые совместно используют общую память. В *распределенной* или *параллельной* компьютерной системе имеется несколько компьютеров (возможно, несколько десятков или сотен), у каждого из которых есть своя память и свой центральный процессор, причем все эти компьютеры при помощи линий связи объединены в одну сеть, через которую они могут взаимодействовать друг с другом. В таких системах одновременно может выполняться множество задач.

Даже если программа выполняется на одном компьютере, часто бывает полезно разработать ее таким образом, чтобы она состояла из нескольких отдельных задач, одновременно выполняющихся на *виртуальном* компьютере, хотя реально на физическом компьютере в каждый момент времени может выполняться только одна задача. Иллюзия параллельного выполнения в случае одного процессора достигается за счет чередования выполнения отдельных задач так, что каждая из них выполняет часть своего кода, после чего процессор переключается на выполнение другой задачи, которая также выполняет только часть своего кода и т. д. Операционные системы, которые поддерживают *мультипрограммирование* и режим *разделения времени*, предоставляют именно такой способ одновременного выполнения нескольких пользовательских программ. Но в нашей книге мы будем иметь дело с параллельным выполнением задач *в пределах одной программы*.

Основным камнем преткновения является то, что в языках программирования отсутствуют конструкции для построения таких систем. По большей части при использовании таких стандартных языков, как C, приходится обращаться к дополнительным функциям операционной системы, чтобы реализовать параллельные задачи. Но ситуацию можно исправить, если разработать соответствующий язык программирования.

Принципы разработки языков параллельного программирования

Конструкции параллельного программирования усложняют разработку языка, поскольку одновременно несколько процессоров могут иметь доступ к одному и то-

му же объекту данных. Чтобы рассматривать параллелизм в языках программирования, следует обратиться к следующим пяти понятиям.

1. *Определение переменных.* Переменные могут быть *изменяемыми* или *неизменяемыми.* Изменяемые переменные — это обычные переменные, объявляемые в большинстве последовательных языков программирования. Им можно присваивать различные значения, которые в процессе выполнения программы могут меняться. Неизменяемой переменной значение можно присвоить только один раз. Смысл существования таких переменных заключается в том, чтобы избежать проблемы синхронизации. Когда такой переменной присвоено некоторое значение, любая задача может обратиться к этой переменной и получить правильное значение.
2. *Параллельная композиция.* Выполнение движется от одного оператора к следующему. Дополнительно к последовательным и условным операторам последовательных языков программирования нам нужно добавить *параллельный* оператор, создающий дополнительный *поток управления,* который начинает выполняться. Примерами таких структур являются оператор and, описанный в разделе 11.2.1, и функция операционной системы fork[1], используемая в языке С.
3. *Структура программы.* Параллельные программы в основном следуют одной из следующих двух моделей выполнения.
   а) Они могут быть *трансформационными,* когда целью является трансформация входных данных в соответствующие выходные значения. Параллелизм применяется для ускорения процесса вычисления, например быстрого перемножения матриц путем параллельного перемножения нескольких ее сечений.
   б) Они могут быть *реагирующими,* когда программа реагирует на некоторые внешние «раздражители», называемые *событиями.* Примерами реагирующих систем являются системы реального времени и оперативные системы управления. Операционная система и система обработки транзакций, например система регистрации предварительных заказов, также являются типичными примерами таких реагирующих систем. В целом их можно охарактеризовать как обладающие недетерминированным поведением, так как никогда не известно точно, когда произойдет какое-то событие.
4. *Взаимодействие.* Параллельные программы должны взаимодействовать между собой. Такое взаимодействие, как правило, осуществляется через *совместно используемую память* с общими объектами данных, доступными для каждой из параллельных программ, или через *сообщения,* когда каждая параллельная программа имеет собственную копию объекта данных и передает значения данных другим параллельным программам.
5. *Синхронизация.* Параллельные программы должны быть способны упорядочивать выполнение своих различных потоков управления. Хотя для мно-

---

[1] По-английски «fork» — вилка, разветвление. — *Примеч. пер.*

гих приложений подходит недетерминированное поведение, некоторым все же требуется определенная упорядоченность. Например, можно создать компилятор, в котором сканер и анализатор будут работать параллельно, но при этом обязательно должно выполняться требование, что сканер считывает каждую лексему до того, как анализатор начнет ее обрабатывать. Описанный выше механизм взаимодействия позволит осуществить это упорядочение.

В следующих подразделах мы обсудим некоторые из этих концепций более подробно.

11.2.1. Параллельное выполнение

Основной механизм, вводящий параллельное выполнение в язык программирования, — это специально созданная конструкция, допускающая такое параллельное выполнение. Оператор and

$оператор_1$ and $оператор_2$ and ... and $оператор_n$

выполняет поставленную задачу, и его семантика заключается в том, что каждый из операторов $оператор_i$ выполняется параллельно. Оператор, следующий за таким оператором and, не начинает выполняться, пока не будут выполнены все параллельные операторы.

Хотя эта конструкция концептуально и проста, она предоставляет всю мощь параллельных вычислений, которую мы хотели бы иметь. Например, если операционная система включает в себя задачу считывания с терминала, задачу вывода информации на экран монитора и процесс выполнения пользовательской программы, мы могли бы определить эту операционную систему следующим образом:

```
call ReadProcess and
call WriteProcess and
call ExecuteUserProgram;
```

Последовательности для параллельного выполнения — это лишь часть проблемы. Корректное управление данными — другая ее часть. Рассмотрим следующий фрагмент:

```
x:=1;
x:=2 and y := x+x; (*)
print(y);
```

Поскольку оба помеченных звездочкой (*) оператора могут выполняться параллельно, мы не можем предсказать, какой из них завершится первым. Следовательно, переменной y может быть присвоено значение 2 (если оператор присваивания значения переменной y выполнится первым), 4 (если оператор присваивания значения переменной x выполнится первым) или даже 3 (если оператор присваивания значения переменной x завершится между двумя обращениями к переменной x в операторе присваивания значения переменной y). Мы должны скоординировать доступ параллельных программ к данным. Эту тему мы обсудим несколько позже при рассмотрении *семафоров* в разделе 11.2.5.

Реализация. Существует два основных способа реализации конструкции and. Заметим, что если все параллельные задачи могут выполняться параллельно, то не делается никаких предположений о порядке их выполнения. Мы можем просто

выполнить их последовательно. Если исходная конструкция and корректна, то замена and на оператор последовательного выполнения «;» и оператор while привела бы к корректному выполнению. Например, приведенный выше пример мог бы быть переписан компилятором как

```
while MoreToDo do
      MoreToDo := false;
      call ReadProcess;
      call WriteProcess;
      call ExecuteUserProgram
end
```

Если бы, например, подпрограмма ReadProcess пыталась прочесть данные, которые еще не готовы для обработки, она могла бы просто установить значение переменной MoreToDo равным true и вернуться назад. Этот цикл будет повторяться до тех пор, пока не завершится каждая из подпрограмм.

Более прямой способ реализовать эту конструкция заключается в использовании элементарных операций операционной системы, предназначенных для организации параллельного выполнения. Например, в C компилятор мог бы выполнить функцию fork, которая создает два выполняемых параллельно процесса. Каждый из этих процессов продолжал бы выполняться, пока не завершится. Код, сгенерированный компилятором C, мог бы выглядеть примерно следующим образом[1]:

```
fork ReadProcess;
fork WriteProcess;
fork ExecuteUserProgram;
wait                              /* ждем, пока не выполнятся все 3 программы */
```

Возможности поддержки параллельных задач еще достаточно редки в языках программирования высокого уровня. Ни один из широко распространенных языков не предоставляет описанную здесь конструкцию and. Только Ada позволяет создавать задачи с параллельным выполнением, хотя тесная связь языка C с операционной системой позволяет C-программам вызывать функцию операционной системы fork для создания параллельных задач.

11.2.2. Охраняемые команды

Второй класс конструкций имеет отношение к недетерминированному выполнению, когда невозможно определить, какой оператор должен будет выполняться следующим. Конструкция and, вообще говоря, является детерминированной, поскольку в следующий момент времени на данном компьютере фактически будет выполняться только один из параллельных операторов. Тем не менее концепция оператора and относится к параллельному выполнению, а ее реализацией часто является последовательный проход через все операторы.

В 1970 г. Дейкстра (Dijkstra) предложил действительно недетерминированную концепцию *охраняемых команд* как средство упрощения, разработки и верифика-

[1] Этот пример только частично корректен, поскольку мы старались сделать его максимально простым. На самом деле компилятор C сначала бы должен был вызвать функцию fork для создания двух процессов, а затем вызвать функцию exec для их выполнения.

ции программ [37]. До сих пор все рассмотренные нами в этой главе структуры управления являются детерминированными. Это означает, что каждая часть оператора имеет определенный порядок выполнения. Но встречаются ситуации, когда недетерминированность позволяет упростить разработку программного обеспечения (см. обсуждение недетерминированного конечного автомата в разделе 3.3.2).

Недетерминированное выполнение — это такое выполнение, при котором возможны несколько альтернативных путей. Такая ситуация часто возникает при выполнении параллельных программ. Если в системе имеется *n* процессов, готовых к выполнению, не всегда понятно, который из них будет выполняться следующим. Хотя при разработке операционных систем эта проблема встречается очень часто, она гораздо реже возникает при разработке отдельной программы, в основном потому, что в нашем распоряжении нет подходящих инструментов (например, операторов языка программирования), позволяющих нам мыслить в терминах недетерминированных вычислений.

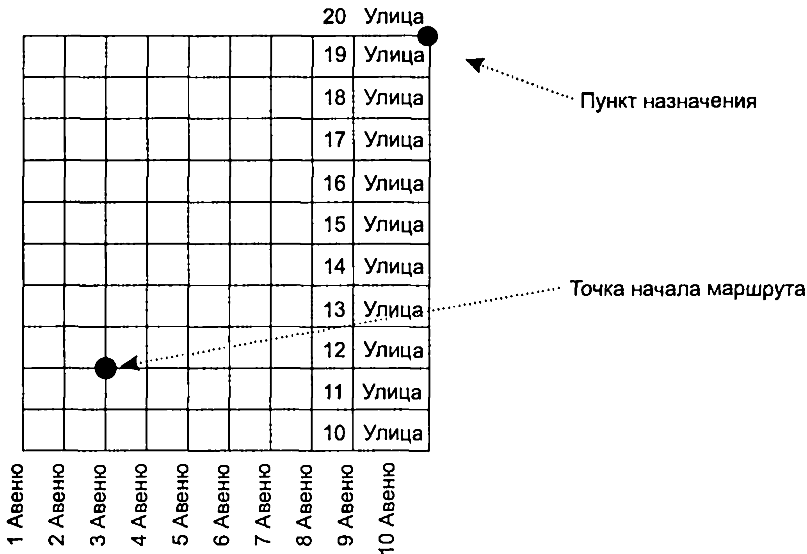


Рис. 11.2. Недетерминированное странствие по городу

В качестве простого примера рассмотрим город, кварталы которого образуются из авеню, идущих в направлении с севера на юг, и улиц, идущих с запада на восток (рис. 11.2). Допустим, вы находитесь на пересечении Третьей авеню и Двенадцатой улицы и вам надо попасть на пересечение Десятой авеню и Двадцатой улицы. Какой путь следует выбрать? Если попросить запрограммировать решение на некотором условном языке, то типичными будут два возможных варианта программ:

Вариант 1	**Вариант 2**
Двигаемся на восток от Третьей авеню до Десятой авеню	Двигаемся на север от Двенадцатой улицы до Двадцатой улицы
Двигаемся на север от Двенадцатой улицы до Двадцатой улицы	Двигаемся на восток от Третьей авеню до Десятой авеню

Но существует и множество других путей. Если придерживаться только двух указанных траекторий, вы можете никогда не увидеть какие-нибудь достопримечательности, находящиеся, например, на Восьмой авеню или на Шестнадцатой улице. Фактически все возможные пути можно описать следующей программой:

```
while не в пункте назначения do arbitrarily
if не на 10-й авеню, иди на восток 1 or
if не на 20-й улице, иди на север 1
end
```

Любой путь, который соответствует этому произвольному (то есть недетерминированному) маршруту, является правильным решением поставленной задачи. Вскоре мы увидим, что охраняемые команды, предложенные Дейкстрой, являются общим решением этой задачи.

Сторожевые условия. Основным понятием является *сторожевое условие* (guard), которое обозначается символом →. Если B является сторожевым условием, а S — *командой* (то есть оператором), то охраняемая команда B → S означает, что оператор S разрешен или готов к выполнению, если сторожевое условие B *истинно.* Мы используем сторожевые условия в *охраняемом* операторе if и *охраняемом* операторе *повторения.*

Охраняемый оператор if. Если B_i — множество условий, а S_i — множество операторов, тогда охраняемый оператор if записывается следующим образом:

```
if B1 → S1 || B2 → S2 || ... || Bn → Sn fi
```

Смысл этого оператора в том, что по крайней мере одно из сторожевых условий *должно* быть истинным, и выполнение этого оператора заключается в выполнении соответствующего оператора.

Обратите внимание на отличие этой конструкции от тех, которые мы уже изучили. Она отличается от стандартной конструкции if-then, так как в обычных случаях не требуется, чтобы сторожевое условие было истинным. Далее, эта конструкция отличается от конструкции cond языка LISP, где выполняемым оператором был первый в последовательности, для которого условие оказывалось истинным. Определение охраняемого оператора if характеризуется истинной недетерминированностью. Мы произвольным образом выбираем любой из операторов S_i, для которого условие B_i истинно.

Охраняемый оператор повторения. Этот оператор является обобщением последовательного оператора while и похож на охраняемый if: Если B_i — множество сторожевых условий, а S_i — множество операторов, то охраняемый оператор do записывается так:

```
do B1 → S1 || B2 → S2 || ... || Bn → Sn od
```

Этот оператор выполняется следующим образом: если некоторые сторожевые условия B_i истинны, то выполняется один из соответствующих операторов S_i. Этот процесс повторяется, пока истинно хотя бы одно сторожевое условие. Если изначально ни одно из сторожевых условий не истинно, то охраняемый оператор do пропускается, подобно тому, как это происходит с обычным оператором while. Как и в случае с охраняемым оператором if, при наличии более чем одного истинного сторожевого условия возникает недетерминированность.

Охраняемые команды часто упрощают разработку и понимание алгоритмов. Вернемся к рассмотренной выше задаче о маршруте по городу. Общее решение может быть задано недетерминированным способом:

```
Ave = 3; {На 3 авеню}
St = 12; {На 12 улице}
do Ave < 10 → Ave = Ave + 1
      || St < 20 → St = St + 1 od
```

Корректным решением, получаемым с помощью этой программы, является любой маршрут, который начинается на пересечении Третьей авеню и Двенадцатой улицы и заканчивается на пересечении Десятой авеню и Двадцатой улицы.

Ни в одном из распространенных языков не реализованы охраняемые команды в том виде, в котором они были предложены Дейкстрой. Тем не менее концепция недетерминированного выполнения действительно возникает при разработке программного обеспечения операционных систем. В разделе 11.2.4 более полно рассматривается роль охраняемых команд при использовании задач и подпрограмм.

11.2.3. Обзор языка Ada

История. Хотя изначально язык Ada разрабатывался для военных приложений, фактически он является универсальным языком и пригоден для решения большинства задач программирования. Ada имеет блочную структуру и похожие на существующие в языке Pascal механизмы описания типов данных, хотя в нем есть расширения для разработки приложений, работающих в режиме реального времени, и распределенных приложений. Он предоставляет более защищенную форму инкапсуляции данных, чем Pascal, а последний пересмотренный стандарт языка Ada был дополнен возможностями разработки объектов и наследования методов.

Язык Ada — уникальный язык, отличающийся от всех остальных, рассмотренных в этой книге, тем, что он стал международным стандартом до того, как появился первый работающий транслятор этого языка. Целью проекта, осуществлявшегося под руководством Министерства обороны США (Department of Defense, DOD), являлась разработка языка и фиксация полученной разработки при помощи стандарта до того, как появились бы несовместимые вариации. До некоторой степени эта цель была достигнута в 1983 г., хотя некоторые недостатки языка привели к пересмотру его стандарта в 1995 г.

В конце 70-х гг. в Министерстве обороны США возник интерес к созданию единого языка для использования во встроенных системах, работающих в режиме реального времени. Была создана Рабочая группа языков высокого порядка (Higher Order Language Working Group, HOLWG) для разработки такого языка, который позволил бы избавиться от царившего в этой области хаоса. В одном из обзоров, составленных группой HOLWG, было перечислено свыше пятисот языков, которые применялись в то время для поддержки различных военных приложений. С помощью ряда последовательных, все более точных спецификаций в период с 1975 по 1978 г. («Strawman», «Woodman», «Tinman», «Ironman» и, наконец, «Steelman») были определены требования для такого языка. Хотя исходной целью было либо использовать, либо расширить какой-нибудь из существующих языков (наиболее вероятными кандидатами являлись Pascal, ALGOL 68 или PL/I), вскоре обнару-

жилось, что ни один из существующих языков не мог отвечать нуждам избранной области приложений и требовался совершенно новый язык.

Чтобы сократить огромный срок (приблизительно 20 лет), который потребовался бы для полной разработки нового языка, был испробован новый подход. Контракты на проектирование такого языка были заключены с несколькими разработчиками одновременно, и результаты этого соревнования проектов стали предметом интенсивного изучения как в промышленности, так и в академической среде с целью доведения сроков экспертных исследований до нескольких месяцев. Из семнадцати представленных вариантов языков было выбрано четыре, которым были присвоены кодовые названия «красный», «зеленый», «желтый» и «синий». После второго отборочного этапа в 1979 г. победителем оказался «зеленый» язык, представленный Жаном Ишбиа (Jean Ichbiah) из Франции. Хотя изначально язык назывался DOD-1, вскоре его имя изменилось на Ada — в честь Ады Лавлейс (Ada Lovelace[1]), пионера в области машинных вычислений и помощницы Чарльза Бэббиджа (Charles Babbage), разработчика первого механического вычислительного устройства в 50-х гг. XIX столетия.

При разработке языка Ada отправной точкой послужил Pascal, но получившийся в результате язык отличается от Pascal во многих основных аспектах. В первую очередь, Ada гораздо более сложный язык, чем Pascal, включает в себя большее количество языковых конструкций и имеет некоторые основные свойства, не имеющие аналогов в Pascal, — в частности, создание задач, параллельное выполнение, управление задачами в режиме реального времени, обработка исключений и возможность создания абстрактных типов данных. Язык был стандартизован в 1983 г. одновременно как коммерческий стандарт США, военный стандарт США и международный стандарт [58].

Чтобы убедиться в том, что трансляторы Ada соответствуют стандарту, в DOD был создан комитет Ada Joint Program Office (AJPO) для наблюдения за применением этого языка в военной области и разработки Ada Compiler Validation Suite (ACVS) — набора тестов, которые должен был пройти любой транслятор Ada, чтобы считаться реализацией языка. Для военных приложений США требовалось, чтобы любой компилятор, используемый в таком проекте, прошел тест ACVS, тогда как для коммерческого или академического использования такой проверки не требовалось.

Использование тестов ACVS было еще одним отличием от обычного процесса стандартизации. Подмножества или расширения стандарта не допускались ни для одного сертифицированного транслятора. Строгое соответствие стандарту требовалось и от содержания справочного руководства: ничего больше и ничего меньше, — хотя имелись и отступления от этого правила. Например, простой процессор, управляющий системой понижения токсичности выхлопа автомобиля, не использовал сложную систему управления файлами. Такие функции не требовались в компиляторах Ada, использовавшихся для приложений в этой области, если целевая система не имела этих возможностей.

Первые эффективные трансляторы Ada появились только в 1986 или 1987 г. Использование этих новых трансляторов, а также академическое эксперименти-

---

[1] Кстати, дочери великого английского поэта лорда Джорджа Гордона Байрона. — *Примеч. науч. ред.*

рование с языком (преподаватели университетов не чувствовали себя связанными такими протоколами, как ACVS) привели к тому, что стали появляться предложения о пересмотре языка или создании расширений к нему. В 1988 г. комитет AJPO вместе с ISO сформировали комитет Ada 9X для пересмотра стандарта. В названии комитета символы «9X» появились в связи с тем, что дата окончания работ не могла быть даже приблизительно оценена, так как подобная работа всегда требует больше времени, чем ожидается. В результате 9X превратились в 95. Помимо решения проблем, связанных со стандартом 1983 г., основные изменения в новом стандарте были связаны с улучшением объектно-ориентированного подхода, иерархических библиотек и модели обработки задач. В этой главе описан язык стандарта 1983 г. и показано, как новый стандарт модифицирует эти возможности.

Краткий обзор языка. Язык Ada предназначен для поддержки создания больших программ большими командами программистов. Программа на языке Ada обычно разрабатывается как набор крупных программных компонентов, называемых *пакетами*, каждый их которых представляет абстрактный тип данных или множество объектов данных, совместно используемых подпрограммами.

Программа на Ada состоит из одной процедуры, играющей роль главной программы. Эта главная программа объявляет переменные и выполняет операторы, в том числе вызовы других подпрограмм. Однако эти типы данных и подпрограммы часто являются частями отдельно определенных пакетов, которые просто импортируются из библиотеки программ по мере надобности. Программа на Ada может вызывать множество отдельных задач, которые должны выполняться параллельно. Если в программе действительно выполняются задачи, то они часто инициируются непосредственно главной программой и образуют верхний уровень структуры программы.

В Ada предусмотрен широкий спектр встроенных типов данных, включая целые и вещественные числа, перечисления, булевы величины, массивы, записи, символьные строки и указатели. Абстракция и инкапсуляции определяемых программистом типов и операций обеспечиваются с помощью свойств пакетов. Механизм инкапсуляции позволяет сделать невидимым внутреннюю структуру объектов данных и подпрограмм, что дает возможность реализовать истинную абстракцию как данных, так и операций.

Управление последовательностью действий внутри подпрограммы использует выражения и структуры управления на уровне операторов, подобные тем, что используются в языке Pascal. Имеется огромный выбор средств для обработки исключительных ситуаций. Управление последовательностью действий на уровне подпрограмм включает обычный вызов подпрограмм и возврат с рекурсией. Кроме того, исключительные ситуации могут приводить к завершению подпрограмм. Но наиболее замечательным аспектом возможностей управления последовательностью действий в Ada является обеспечение выполнения параллельных задач и управление ими с использованием системных часов и других механизмов планирования.

Структуры управления данными в Ada используют статическую блочно-структурированную организацию, как в языке Pascal. Однако в языке также предусмотрены средства для нелокальных ссылок на имена типов, имена подпрограмм и другие идентификаторы в явной общей среде, определяемой пакетом, в которой часто

один и тот же идентификатор может быть использован во многих различных пакетах. Каждый формальный параметр в подпрограмме помечается ключевым словом in, out или in out.

Виртуальный компьютер Ada использует центральный стек для каждой отдельной задачи. Каждый стек используется для создания и уничтожения записей активации подпрограмм во время выполнения этой задачи. Поскольку задачи могут выполняться параллельно, они могут выполнять подпрограммы независимо друг от друга. Для объектов данных, создаваемых программистом, также требуется область памяти, отводимая под кучу. Так как отдельные задачи в Ada-программе могут потенциально выполняться на отдельных компьютерах, то и области памяти, используемые каждой задачей, могут быть распределены в памяти разных компьютеров. Такая распределенная природа хранения данных и выполнения задач порождает специфические проблемы в реализации Ada, которых не найти ни в одном из остальных описанных в этой книге языков.

11.2.4. Задачи

Идея, лежащая в основе задач, достаточно проста. Рассмотрим подпрограмму A, которая выполняется обычным образом. Если A вызывает подпрограмму B, то обычно выполнение A приостанавливается на то время, пока выполняется B. Но если B инициирована как *задача*, то A продолжает выполняться одновременно с B. Исходная последовательность выполнения теперь разделилась на две параллельных последовательности выполнения. Теперь или подпрограмма A, или подпрограмма B, или обе могут инициировать другие задачи, позволяя любому числу параллельных последовательностей выполнения сосуществовать одновременно.

Вообще говоря, каждая задача является *зависимой* от той задачи, которая ее инициировала. Когда задача готова к завершению, она должна дождаться, пока завершатся все зависимые от нее задачи. Таким образом, разделенная на несколько последовательностей выполнения первоначальная последовательность по мере завершения задач становится состоящей из все меньшего числа параллельных последовательностей, пока снова, после завершения последней зависимой задачи, не останется одна последовательность выполнения. При нормальных обстоятельствах каждая из этих задач верхнего уровня управляет большей частью системы (в настоящее время часто распределенной компьютерной системы); ожидается, что, однажды инициированные, они будут выполняться нескончаемо долго.

Управление задачами

Определение задачи в программе мало отличается от определения обычной подпрограммы за исключением определения того, как задача синхронизируется и взаимодействует с другими задачами. Большая часть тела определения задачи состоит из обычных объявлений и операторов, реализующих функциональность задачи, пока она работает независимо от других задач. В языке Ada, который будет здесь нашим основным демонстрационным языком, определение задачи принимает следующий вид:

```
task Name is
        -Объявления для синхронизации и взаимодействия с другими задачами
```

```
    end;
task body Name is
        -Обычные локальные объявления, свойственные любой подпрограмме
    begin  -Последовательность операторов
    end;
```

Инициализация выполнения задачи может принимать вид обычного вызова подпрограммы. Например, во многих реализациях PL/I задача B инициируется выполнением следующего оператора:

```
call B(параметры) task;
```

В Ada используется несколько иной метод. Определение задачи, как сказано ранее, включается в набор объявлений некоторой более крупной программной структуры, такой, например, как главная программа. При входе в эту более крупную программную структуру все объявленные в ней задачи автоматически инициируются. Таким образом, не нужен явный оператор вызова; задачи начинают выполняться параллельно сразу после входа в содержащую их большую программную структуру.

В приложениях часто требуется несколько одновременных активаций *одной и той же* задачи. Рассмотрим, например, компьютерную систему, которая управляет некоторым множеством пользовательских терминалов. Основной задачей может являться программа, которая следит за состоянием всех мониторов. Когда какой-либо пользователь входит в систему с некоторого терминала, эта задача, Monitor, инициирует новую задачу, Terminal, чтобы управлять взаимодействием с пользователем, работающим за этим конкретным терминалом. Когда пользователь выходит из системы, задача Terminal завершается. Задача Monitor, конечно, работает непрерывно за исключением аварийной остановки всей системы. Когда несколько пользователей входят одновременно в систему с различных терминалов, то требуется несколько активаций задачи Terminal, по одному на каждого пользователя.

Если задача инициирована с использованием обычного оператора вызова подпрограмм, как в PL/I, тогда повторного выполнения оператора вызова достаточно для создания нескольких активаций. В языке Ada приведенное выше определение задачи может быть использовано только для создания одной активации задачи в соответствии с правилами неявной инициализации задач. Поэтому задача Monitor будет, вероятно, определена, как и ранее. Для задачи Terminal требуется несколько активаций, и они должны создаваться и инициироваться задачей Monitor по мере необходимости. В Ada задача Terminal определяется как *тип* task:

```
task type Terminal is
      -остальная часть определения совпадает с обычным определением
end;
```

Определение Terminal как тип task позволяет рассматривать активацию задачи как тип объекта данных в том же смысле, как обычное определение типа используется для определения класса объектов данных (см. раздел 6.3). Создание и инициирование новой активации задачи тогда аналогично созданию нового объекта данных с использованием определения типа в качестве шаблона. Для создания нескольких активаций с именами A, B и C программист на языке Ada пишет соответствующие объявления так же, как объявления обычных переменных:

```
A: Terminal;
B,C: Terminal;
```

Эти объявления появляются в начале более крупной программной структуры; при входе в нее создаются и инициируются три активации задачи типа Terminal. Другой способ заключается в том, чтобы определить переменную-указатель, значением которой является указатель на *активацию задачи*, как в следующем фрагменте:

```
type TaskPtr is access Terminal:        -Определение типа указатель
NewTerm: TaskPtr := new Terminal:       -Объявление переменной типа указатель
```

Переменная NewTerm указывает на активацию задачи типа Terminal, которая создается одновременно с созданием переменной NewTerm.

Когда задача инициирована, операторы ее тела выполняются последовательно, так же как и в обычной подпрограмме. Когда задача завершается, она не возвращает управление; просто завершается ее отдельная, параллельная последовательность выполнения. Но задача не может быть завершена, пока не завершилось выполнение зависимых от нее задач; когда задача завершается, любая задача, для которой она является зависимой, должна быть проинформирована об этом, чтобы она также могла завершиться. Задача завершается, когда выполнены все содержащиеся в ее теле операторы; задача, которая никогда не завершается, содержит в своем теле бесконечный цикл, который непрерывно выполняется (пока не произойдет какая-либо ошибка).

11.2.5. Синхронизация задач

Когда несколько задач выполняется параллельно, каждая из этих задач выполняется асинхронно по отношению к другим, то есть каждая задача выполняется со своей собственной скоростью, независимой от скорости выполнения других задач. Например, если в задаче A выполнено 10 операторов, то в параллельной задаче B, инициированной в тот же самый момент времени, может быть выполнено 6 операторов, или ни одного, или она вообще может уже быть выполнена и находиться в процессе завершения.

Для координации деятельности двух асинхронно выполняющихся задач в языке должны быть предусмотрены средства *синхронизации*, чтобы одна задача могла сообщить другой, в какой момент завершается выполнение определенной части ее кода. Например, одна задача может контролировать входное устройство, а вторая задача — обрабатывать каждый пакет данных по мере его получения с устройства ввода. Первая задача читает пакет данных, сигнализирует второй задаче, что пакет данных уже поступил, и затем начинает подготовку к получению следующего пакета данных. Вторая задача ожидает сигнала от первой задачи, обрабатывает данные, сигнализирует первой, что обработка данных закончена, и затем ожидает поступления следующего сигнала о получении очередного пакета данных. Сигналы, пересылаемые между задачами, позволяют им синхронизировать свою деятельность, так что вторая задача не начинает обработку данных, пока первая не завершит их чтение, и первая задача не перезаписывает данные, которые вторая еще обрабатывает.

Задачи, которые синхронизируют свою работу описанным способом, в некотором смысле похожи на сопрограммы. Сигналы служат для сообщений каждой задаче, когда ей следует выполнять обработку, а когда нужно просто ждать, — это

напоминает использование вызовов возобновления выполнения (resume) сопрограммами, когда одна сопрограмма сигнализирует другой о том, что та может продолжить свое выполнение. Однако отличие заключается в том, что в случае сопрограмм существует одна последовательность выполнения, в то время как в случае с задачами последовательностей выполнения может быть несколько.

Прерывания. Синхронизация параллельных задач с помощью прерываний — это широко распространенный механизм, встроенный в аппаратную часть компьютера. Если задаче `A` требуется сообщить задаче `B`, что произошло определенное событие (например, выполнился определенный сегмент кода), то задача `A` выполняет команду, следствием которой является немедленное прерывание задачи `B`. Управление передается подпрограмме или сегменту кода, единственное назначение которого — обработать прерывание, выполнив все специальные требуемые действия. Когда этот обработчик прерывания завершает свою работу, выполнение задачи `B` возобновляется с той точки, на которой произошло прерывание. Этот метод сигнализации аналогичен механизмам обработки исключений, описанным в разделе 11.1.1, и часто используется для этих целей. Например, в аппаратной части компьютера задача, контролирующая устройство ввода-вывода, может синхронизироваться с центральным процессором с помощью прерываний. В языках высокого уровня, однако, использование прерываний как механизма синхронизации имеет ряд недостатков:

1) код для обработки прерываний отделен от основного тела задачи, что приводит к путаной структуре программы;
2) задача, которая ожидает, когда произойдет прерывание, обычно должна войти в *цикл активного ожидания* — цикл, который ничего не вычисляет, а просто непрерывно выполняется, пока не произойдет прерывание;
3) задача должна быть составлена таким образом, чтобы прерывание могло быть корректно обработано в любой момент времени, для чего обычно требуется, чтобы данные, совместно используемые операторами в теле задачи и программой обработки прерывания, были специальным образом защищены.

Из-за этих (и некоторых других) проблем с прерываниями в языках высокого уровня обычно используются другие механизмы синхронизации.

Семафоры. *Семафор* — это объект данных, который используется для синхронизации задач. Семафор состоит из двух частей:

1) целочисленного счетчика, значением которого может быть ноль или положительное число, означающее количество посланных, но еще не полученных сигналов,
2) очереди задач, которые ждут сигнала.

В *двоичном семафоре* значение счетчика всегда либо 0, либо 1. В *общем семафоре* счетчик может принимать любое неотрицательное значение.

Для объекта данных *семафор* `P` определены две элементарные операции:

✦ `signal(P)`. Когда эта операция выполняется задачей `A`, то она проверяет значение счетчика в `P`; если оно равно 0, то первая задача из очереди задач удаляется и ее выполнение возобновляется; если значение не равно 0 или очередь пуста, то счетчик увеличивается на 1 (это означает, что сигнал был

послан, но еще не получен). В любом случае выполнение задачи A продолжается после того, как операция signal завершена.

- wait(P). Когда эта операция выполняется задачей B, то она проверяет значение счетчика в P; если оно отлично от нуля, то значение счетчика уменьшается на 1 (означая, что задача B получила сигнал) и задача B продолжает выполняться; если значение счетчика равно 0, то задача B вставляется в конец очереди задач для семафора P и ее выполнение приостанавливается (это означает, что задача B ждет сигнала).

Семантика операций signal и wait проста и основана на принципе *атомарности*. Каждая операция полностью завершается, прежде чем любая другая параллельная операция сможет получить доступ к ее данным. Атомарность предотвращает возникновение определенных классов нежелательных недетерминированных событий. Например, если через atom(S) мы обозначим атомарное выполнение (то есть выполнение, которое не может быть прервано) оператора S, то недетерминированный пример из раздела 11.2.1 можно переписать следующим образом:

```
x:=1;
x:=2 and atom(y := x+x);
print(y);
```

В этом случае результат может быть или 2, или 4 в зависимости от того, какой оператор присваивания выполнится в первую очередь, но в любом случае мы не получим нежелательного результата 3.

В качестве примера использования семафоров и операций signal и wait рассмотрим вновь две задачи, которые взаимодействуют:

1) при получении пакета данных (задача A);
2) при обработке этого пакета (задача B).

Для синхронизации их действий можно было бы использовать два двоичных семафора. Семафор StartB используется задачей A для того, чтобы сигнализировать о завершении приема пакета данных. Семафор StartA используется задачей B для того, чтобы сообщать о завершении обработки данных. В листинге 11.1 показана структура этих задач, использующих семафоры.

Листинг 11.1. Синхронизация задач с помощью операций signal и wait

```
task A;
begin
   - Ввод первого пакета данных
   loop
      signal(StartB)   - Вызов задачи B
      - Ввод следующего пакета данных
      wait(StartA)     - Ожидание завершения обработки данных задачей B
   endloop;
end A;
task B;
begin
   loop
      wait(StartB)     - Ожидание завершения приема данных задачей A
      - Обработка данных
      signal(StartA)   - Сигнал задаче A, что она может продолжать работу
   endloop;
end B;
```

При программировании задач на языках высокого уровня семафоры имеют некоторые недостатки:

1) задача может ждать сигнала только от одного семафора за один раз, однако часто требуется, чтобы задача могла ожидать нескольких различных сигналов;
2) если задача не смогла выдать сигнал в нужной точке (например, из-за ошибки в коде), то целая система задач может *зависнуть* (то есть каждая задача в очереди семафора может ожидать, чтобы какая-либо другая задача выдала сигнал, так что ни одна задача не остается в состоянии выполнения);
3) программы, содержащие несколько задач и семафоров, становятся значительно труднее для понимания, отладки и верификации.

В сущности, семафор — это конструкция синхронизации относительно низкого уровня, которая пригодна лишь в самых простых ситуациях.

В современных операционных средах проявляется еще один недостаток семафоров. Семантика операций signal и wait подразумевает, что все задачи, имеющие доступ к семафору, совместно используют одну и ту же область памяти. С расширением использования мультипроцессорных систем и вычислительных сетей это предположение перестает быть верным. Использование *сообщений* похоже на использование семафоров, но лишено ограничений, связанных с совместным использованием пространства памяти.

Сообщения. Сообщение — это передача информации от одной задачи к другой. Сообщения позволяют задачам синхронизировать свои действия с другими задачами, однако при этом у задачи остается возможность продолжать свое выполнение, когда синхронизация не требуется. Основная концепция похожа на концепцию *программного канала*: сообщение помещается в канал (или *очередь сообщений*) при помощи специальной команды *посылки сообщения* (send), тогда как задача, ожидающая сообщение, с помощью команды *получения сообщения* (receive) получает доступ к нему с другого конца программного канала. Задача, посылающая сообщение, может продолжать свое выполнение, послать еще какие-либо сообщения и поставить их в очередь, тогда как задача, принимающая сообщение, будет продолжать выполняться, пока в очереди имеются сообщения, требующие обработки.

Например, типичным приложением, где могут использоваться сообщения, является приложение для решения задачи *производитель-потребитель*. Задача-производитель получает новые данные (например, считывает данные, введенные с клавиатуры), а задача-потребитель обрабатывает эти данные (например, компилирует вводимую программу). Если send(to, message) будет обозначать, что задача посылает сообщение message задаче с именем to, а receive(from, message) будет обозначать, что задача будет ожидать получения сообщения message от задачи с именем from, то задачу-производитель можно запрограммировать следующим образом:

```
task Producer;
begin
   loop   - цикл выполняется, пока есть данные для считывания
          - считывание новых данных;
      send(Consumer,data)
   endloop;
end Producer
```

а задачу-потребитель — так:

```
task Consumer;
begin
   loop  - цикл выполняется. пока есть данные для обработки
      receive(Producer.data);
         - Обработка новых данных
   endloop;
end Consumer
```

Сообщения могут быть достаточно разнообразными. Задача 1 в конце главы посвящена исследованию того, как можно использовать сообщения для моделирования семафоров.

Реализация передачи сообщений сложнее, чем может показаться на первый взгляд. Несколько задач могут одновременно попытаться отправить сообщения одной и той же задаче. Если нельзя допустить, чтобы какое-то из этих сообщений потерялось, то в реализации языка должен быть предусмотрен механизм для хранения этих сообщений в очереди (которая обычно называется *буфером*), пока принимающая задача сможет их обработать. Другой вариант заключается в том, чтобы задача, посылающая сообщение (а не ее сообщение), могла ожидать в очереди, пока принимающая задача будет готова к приему сообщения. Этот способ используется в языке Ada, в котором посылающая сообщение задача должна организовать рандеву с принимающей сообщение задачей (и таким образом синхронизироваться с ней), прежде чем сообщение может быть передано.

Охраняемые команды. В разделе 11.2.2 мы определили охраняемые команды как способ введения недетерминированности в программирование. Синхронизация является одной из форм недетерминированности, поскольку не всегда ясно, какая задача будет выполняться следующей. Охраняемые команды создают хорошую модель для синхронизации задач.

Охраняемая команда if в языке Ada называется оператором select и имеет следующую форму (в общем виде, так как в Ada имеются дополнительные ограничения, которые здесь не упомянуты):

```
select
   when условие1 => оператор1
   or when условие2 => оператор2
   ...
   or when условиеn => операторn
   else операторn+1   - необязательное предложение else
   end select;
```

Как и в случае охраняемого оператора if, каждое из условий называется *сторожевым условием*, а каждый оператор — *командой*. Одно из сторожевых условий, которое оказывается истинным, определяет следующий выполняемый оператор. Хотя охраняемые команды могут использоваться как часть разнообразных механизмов синхронизации задач, их использование в механизме рандеву языка Ada хорошо иллюстрирует подобный подход.

Рандеву. В Ada используется метод синхронизации, очень похожий на метод сообщений, но требующий установки синхронизации для каждого сообщения. Когда две задачи синхронизируют свои действия на короткий период времени, такая синхронизация в Ada называется *рандеву*. Предположим, что одна задача A используется для ввода данных, как и в предыдущем примере, а вторая задача B обраба-

тывает эти данные. Предположим также, что B *копирует* получаемые от задачи A данные в свою локальную область, прежде чем начать их обработку, так что A может читать новый пакет данных, не дожидаясь обработки предыдущего. В таком случае необходимо организовать рандеву, чтобы позволить A послать сигнал B о готовности нового пакета данных. Задача A затем должна ждать, пока B скопирует эти данные в свою локальную область, а затем обе задачи могут продолжать выполняться параллельно до тех пор, пока A не завершит чтение нового пакета данных и B не завершит обработку последнего полученного ею пакета, после чего организуется новое рандеву.

Точка рандеву в B называется *точкой входа*, которая в нашем примере может быть названа DataReady. Когда задача B готова начать обработку нового пакета данных, она должна выполнить оператор accept:

```
accept DataReady do
        - операторы копирования новых данных от задачи A в локальную область задачи B
end;
```

Когда задача A завершает ввод нового пакета данных, она должна выполнить обращение к точке входа DataReady. Когда выполнение задачи B доходит до оператора accept, она ожидает, пока задача A (или какая-либо другая задача) выполнит обращение к точке входа DataReady, указанной в операторе accept. Аналогично, когда выполнение задачи A доходит до обращения к точке входа DataReady, она ожидает, пока B дойдет до выполнения оператора accept. Когда обе задачи достигли указанных точек, то происходит рандеву: A продолжает находиться в состоянии ожидания, пока B выполняет все операторы, содержащиеся внутри конструкции do ... end оператора accept. После их выполнения рандеву завершается, и обе задачи A и B продолжают выполняться параллельно.

Чтобы увидеть, как охраняемые команды могут быть использованы для организации ожидания задачей B любого из нескольких рандеву, давайте предположим, что задача B расширена таким образом, что может обрабатывать данные из трех различных входных устройств, каждое из которых управляется отдельной задачей — A1, A2 и A3. Каждая из этих задач выполняется параллельно с B и друг с другом. Когда у одной из задач, получающих данные из устройства ввода, имеется очередной пакет данных, готовый к обработке, она выполняет обращение к соответствующей точке входа — Ready1 (в задаче A1), Ready2 (в задаче A2) или Ready3 (в задаче A3). Когда выполняется обращение к одной из этих точек входа, задача B может уже находиться в состоянии ожидания или еще выполнять обработку ранее полученного пакета данных. Если бы задача B уже находилась в состоянии ожидания, тогда без структуры охраняемых команд она не смогла бы ожидать обращения к *любой* из трех точек входа — Ready1, Ready2 или Ready3, но вместо этого должна была бы ожидать обращения только к одной из них. Чтобы ожидать обращения к любой одной из трех точек входа, задача B выполняет следующую охраняемую команду:

```
select accept Ready1 do
    - Копирование данных от A1 в локальную область B
    end;
or accept Ready2 do
    - Копирование данных от A2 в локальную область B
```

```
        end:
    or accept Ready3 do
        - Копирование данных от A3 в локальную область B
        end;
    end select:
```

Когда выполнение B доходит до этого оператора, программа ожидает, пока A1, A2 или A3 не сигнализируют об обращении к соответствующей точке входа. Это обращение принимается (если одновременно поступило несколько сигналов, то принимается только один из них), и рандеву происходит в соответствии с описанным выше алгоритмом. Заметим, что явные сторожевые команды when условие => и блок else здесь опущены, так как все три оператора accept должны быть доступны для выполнения, когда доходит очередь до выполнения оператора select. В некоторых случаях, однако, явные сторожевые команды могут быть и включены. Например, каждое устройство ввода может иметь соответствующий статус состояния, указывающий, есть или нет сбои в его работе. Тогда рандеву может быть поставлено в зависимость от состояния каждого устройства:

```
select
      when Device1Status = ON => accept Ready1 do ... end ;
or when Device2Status = ON => accept Ready2 do ... end ;
or when Device3Status = connected => accept Ready3
          do ... end ;
else ...   - ни одно из устройств ввода не готово: выполняем другие действия
end select;
```

Задачи и вычисления в режиме реального времени

Программа, которая должна взаимодействовать с устройствами ввода-вывода или другими задачами в течение некоторого фиксированного промежутка времени, называется работающей в *режиме реального времени*. Программам, осуществляющим мониторинг работы автомобиля, управляющим микроволновой печью и даже электронными часами, для корректного функционирования приходится своевременно реагировать за время менее 100 мс. Так, например, задача A, которая должна организовать рандеву с задачей B, возможно, не имеет права задерживать свое выполнение на период времени более некоторого фиксированного значения, а обязана продолжить его, даже без начала выполнения рандеву. В компьютерных системах реального времени сбой в работе аппаратуры устройства ввода-вывода часто приводит к тому, что задача неожиданно завершается. Если другие задачи ожидают какого-либо сигнала от этой завершившейся задачи, вся система связанных между собой задач может зависнуть и привести к полному отказу системы.

Специальные требования, предъявляемые к вычислениям в режиме реального времени, приводят к тому, что язык программирования должен включать некоторое явное понятие *времени*. В Ada существует определенный в языке пакет, называемый Calendar, который включает тип Time (время) и функцию clock (часы). Задача, ожидающая рандеву, может использовать часы, как в следующем фрагменте кода:

```
select DataReady:
or delay 0.5              - ждать не более 0.5 секунды
end select:
```

Это — охраняемая команда без каких-либо сторожевых команд, поэтому любая из альтернатив доступна для выполнения, что позволяет избежать краха системы, если в течение 0,5 с не произойдет обращения к точке входа `DataReady`.

Задачи и совместно используемые данные

Каждая отдельная обработка любой из многочисленных структур управления подпрограммами, описанных в разделе 11.1 (например, сопрограммы, обработчики исключений, задачи и планируемые подпрограммы), приводит к несколько различающимся структурам для совместно используемых данных. В большинстве случаев эти структуры являются простыми вариациями концепций, рассмотренных в предыдущих разделах, но задачи из-за параллельности их выполнения порождают специфические проблемы. Здесь следует рассмотреть два вопроса:

1) управление памятью;
2) взаимное исключение доступа.

Управление памятью в задачах. Задачи определяются как несколько последовательностей выполнения, выполняющихся одновременно в пределах одной программы. Обычно они представляют собой несколько наборов процедур, которые выполняются независимо и взаимодействуют между собой с помощью механизмов взаимного исключения доступа, описываемых ниже. Для каждой задачи требуется свой механизм управления памятью — как правило, для этой цели используется стек. Поскольку каждая задача выполняется независимо, разработчику реализации языка приходится решать непростую задачу: как реализовать несколько стеков на одном компьютере.

На рис. 11.3 представлено несколько способов решения этой задачи. На рис. 11.3, *а* проиллюстрирован обычный способ управления памятью для одной задачи, применяемый в языках типа Pascal и C, в которых стек и куча растут с противоположных концов основной памяти; если стек и куча в процессе этого роста встречаются, то программа вынужденно останавливается, поскольку не остается свободной памяти. Все пространство памяти используется очень эффективно, хотя, как сказано в разделе 10.4.3, некоторая часть памяти может оставаться неиспользованной из-за ее фрагментации в куче.

В случае, если памяти заведомо достаточно, можно применить схему, изображенную на рис. 11.3, *б*. Каждая задача имеет свой собственный стек в отдельном месте памяти. Опять-таки, в случае перекрытия любым стеком сегмента памяти, выделенного под следующий стек, программа будет вынуждена остановиться. С учетом использования современных систем виртуальной памяти это решение достаточно эффективно. Например, адресное пространство программы может состоять из миллиарда адресов ячеек, но из них только несколько тысяч фактически используемых будут находиться в реальной памяти. Если стеки различных задач будут достаточно далеко расположены друг от друга в виртуальной памяти (например, адрес первой ячейки в стеке задачи 1 будет 100 000 000, в стеке задачи 2 — 200 000 000, в стеке задачи 3 — 300 000 000, в куче — 400 000 000), то вероятность их перекрывания достаточно мала. Таким образом, для управления этими несколькими стеками можно использовать систему управления памятью операционной системы, в которой выполняются задачи, при этом потребуется лишь незначительная помощь со стороны транслятора языка помимо установления начальных адресов стеков.

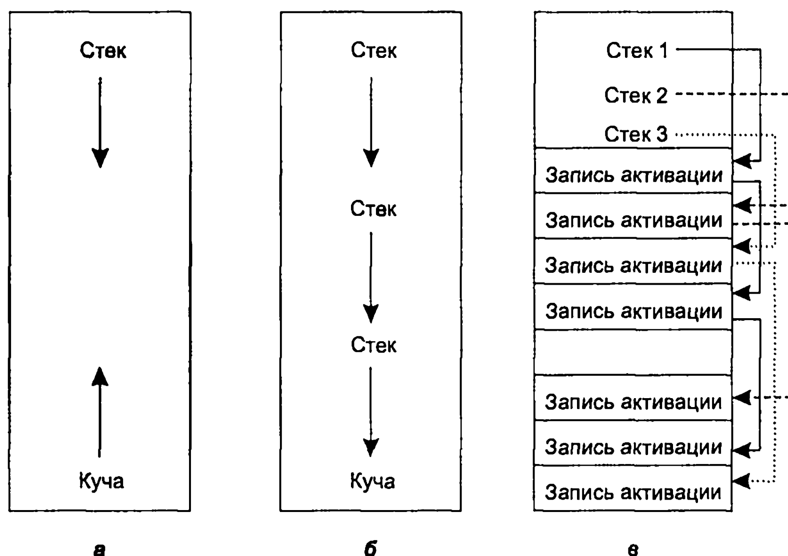


Рис. 11.3. Записи активаций в среде задач: а — один стек; *б* — несколько стеков; *в* — одна куча

Для систем с ограниченными ресурсами памяти (например, если отсутствует система виртуальной памяти или адресное пространство виртуальной памяти сравнительно мало) рассмотренный выше способ может оказаться неприменимым. Тогда можно использовать схему управления памятью, представленную на рис. 11.3, *в*. В этом случае вся память организована в виде кучи и каждый стек состоит из записей активации, расположенных в куче и связанных между собой. Этот механизм работает всегда. В частности, он был использован в ранних компиляторах языка PL/I. Однако этот способ управления памятью связан с большими издержками. Все активации процедур и возвраты из них осуществляются с помощью требующих времени обращений к системной программе распределения памяти, а не с помощью эффективной стековой модели. Если каждая запись активации имеет свой собственный уникальный размер, то фрагментация памяти, которая не встречается в стековой модели, становится здесь критической проблемой. Сравнительный анализ временных затрат показывает, что, как правило, существенная часть всего времени работы программы тратится на выполнение системных подпрограмм распределения памяти и программист практически ничего не может сделать для ускорения работы такой программы. Поэтому применять описанный здесь третий способ управления памятью, вообще говоря, нежелательно, хотя иногда другого способа просто не остается (если требуется реализовать несколько независимо выполняющихся задач, а адресное пространство памяти недостаточно велико).

Поскольку любая задача требует наличия своего собственного центрального стека для хранения записей активации вызываемых ею подпрограмм, выполнение программы начинается с того, что создается единственный стек для главной процедуры. По мере инициирования задач каждой из них требуется выделять область для размещения нового стека. Тогда исходный стек делится на несколько новых стеков, которые, в свою очередь, также могут разделяться, если инициируются но-

вые задачи. Такая структура управления памятью известна как *стек-кактус*, поскольку ее схематическое изображение (рис. 11.4) напоминает очертания гигантского кактуса цереус. Соединение стека вновь инициированной задачи со стеком той программной единицы, в которую эта задача статически вложена, должно поддерживаться во время выполнения программы, поскольку задача может иметь нелокальные ссылки на совместно используемые данные в стеке программной единицы. Таким образом, должна поддерживаться некоторая связь, например указатель статической цепочки, которая позволит правильно разрешить эти нелокальные ссылки во время выполнения программы.

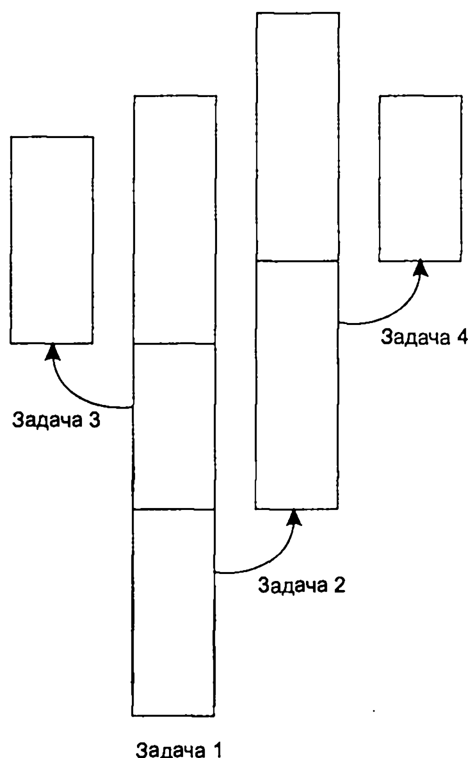


Рис. 11.4. Модель стека-кактуса для нескольких задач

Взаимное исключение. Если две задачи A и B имеют доступ к одному и тому же объекту данных X, то они должны синхронизировать свой доступ к X таким образом, чтобы задача A не могла присвоить новое значение объекту данных X в тот момент, когда задача B ссылается на него или присваивает ему другое значение. Например, если значение переменной X равно 1 и задача A выполняет оператор

```
if X > 0 then X := X + 1;
```

а B выполняет оператор

```
if X > 0 then X := X - 2;
```

то окончательное значение X может оказаться либо 0 (если сначала выполняется A), либо -1 (если сначала выполняется B), либо, возможно, 2 (если случится так, что

A и B будут чередовать свои действия в попытке выполнить оба оператора одновременно). Чтобы можно было гарантировать, что две задачи не станут предпринимать одновременных попыток доступа к совместно используемому объекту данных, одна из них должна обладать *исключительным правом доступа* к объекту данных во время его обработки. Существует несколько различных способов решения проблемы взаимного исключения, когда задачи работают с совместно используемыми данными. В разделе 11.2 мы уже обсуждали семафоры и атомарность, которые можно использовать для синхронизации данных. Далее будут рассмотрены другие механизмы.

Критические области

Критическая область — это последовательность операторов в задаче, в которой осуществляется обработка некоторого объекта данных, используемого совместно с другими задачами. Если критическая область задачи A обрабатывает объект данных X, то принцип внутреннего исключения требует, чтобы никакая другая задача не выполняла одновременно с A свою критическую область, в которой обрабатывается тот же самый объект X. Во время выполнения задачи A, когда она собирается начать выполнение критической области, ей придется ждать, пока любая другая задача завершит выполнение своей критической области, обрабатывающей объект данных X. Как только задача A начинает выполнение своей критической области, все остальные задачи должны быть заблокированы, чтобы они не могли войти в свои критические области (для переменной X), пока A не закончит выполнение своей критической области. Критические области в задачах можно реализовать посредством связывания семафора с каждым совместно используемым объектом данных (или группой объектов). Совместно используемые объекты данных обычно являются частью явной общей среды (или нескольких общих сред), доступной для каждой задачи.

Монитор

Другим способом реализации взаимного исключения является использование *монитора*. Монитор — это совместно используемый объект данных вместе со множеством операций, которые могут им манипулировать. Таким образом, монитор аналогичен объекту данных, определяемому при помощи абстрактного типа данных, как описано в разделе 6.2. Задача может манипулировать совместно используемым объектом данных только с помощью определенных операций, так что объект данных является инкапсулированным, как это обычно и происходит с объектами данных, определяемыми с использованием абстрактных типов данных. Для того чтобы реализовать взаимное исключение, необходимо лишь потребовать, чтобы в любой момент времени можно было выполнять не более одной операции, определенной для этого объекта данных.

Требования для взаимного исключения и инкапсуляции в мониторе позволяют естественным образом представить его в виде задачи. Совместно используемый объект данных представляется локальным объектом данных для этой задачи, а операции определяются как локальные подпрограммы задачи. К примеру, предположим, что совместно используемый объект данных — это таблица BigTable и для нее определены две операции, EnterNewItem и FindItem. Для того чтобы одна из задач не пыталась изменить значение некоторого элемента в таблице в тот момент, когда

другая задача пытается получить значение этого же элемента, необходимо взаимное исключение. В Ada монитор может быть представлен как задача TableManager с двумя точками входа, EnterNewItem и FindItem (листинг 11.2). В пределах этой задачи BigTable является локальной переменной. Оператор select с двумя вариантами accept используется для того, чтобы позволить монитору отвечать на запросы других задач на выполнение одной из двух операций, EnterNewItem или FindItem, таким образом, что в любой момент времени будет выполняться только одна из них. Двум задачам может одновременно потребоваться ввод или поиск каких-либо элементов таблицы. Например, задача A может выполнить оператор обращения к одной точке входа

```
EnterNewItem(...);
```

а задача B может в тот же момент обратиться к другой точке входа:

```
FindItem(...);
```

Первое, полученное задачей TableManager обращение к точке входа начинает обрабатываться (то есть происходит рандеву, как описано в разделе 11.2). Если второй оператор обращения к точке входа выполняется до того, как задачей TableManager обработан первый, то вторая задача должна дождаться окончания его обработки. Таким образом, таблица BigTable защищена от одновременного доступа из двух различных задач.

Листинг 11.2. Представление монитора в виде задачи Ada

```
task TableManager is
   entry EnterNewItem(...);
   entry FindItem(...);
end;
task body TableManager is
   BigTable: array (...) of
   procedure Enter(...) is
         - Операторы для ввода элемента в BigTable
   end Enter;
   function Find(...) returns ... is
         - Операторы для поиска элемента в BigTable
   end Find;
begin
         - Операторы для инициализации BigTable
   loop  - Бесконечный цикл для обработки точек входа
      select
         accept EnterNewItem(...) do
         - Вызов Enter для ввода полученного элемента в BigTable
         end;
      or accept FindItem(...) do
         - Вызов Find для поиска запрошенного элемента в BigTable
         end;
      end select;
   end loop;
end TableManager;
```

Передача сообщений

Другим решением проблемы совместного использования данных задачами является запрещение использования общих *объектов* данных, а предоставление использования только *значений* таких объектов — путем передачи их в виде сообще-

ний. Это та же самая концепция сообщений, которая уже обсуждалась ранее в связи с синхронизацией задач. Использование передачи сообщений в качестве основы совместного использования данных гарантирует взаимное исключение без применения какого-либо специального механизма, поскольку каждый объект данных принадлежит в точности только одной задаче и никакая другая задача не может получить к нему непосредственный доступ. Если некий объект данных принадлежит задаче A, то она посылает копию значений, хранящихся в этом объекте данных, на обработку задаче B. Теперь в B имеется своя локальная копия этого объекта данных. Когда B завершит обработку своей локальной копии этого объекта, она перешлет копию с новыми значениями обратно в A, и затем A изменит фактический объект данных. Разумеется, задача A может продолжать изменение фактического объекта данных, пока B изменяет свою локальную копию.

11.3. Развитие аппаратной части компьютера

Одним из способов, позволяющих решать на компьютере более сложные задачи, является увеличение скорости его работы. Хотя в настоящее время компьютеры становятся все более быстродействующими (приблизительно каждые два года скорость увеличивается вдвое), основная проблема все же остается. Память устройств управления и центрального процессора функционирует примерно на порядок быстрее, чем основная оперативная память большого размера. Следовательно, большую часть времени компьютер проводит в ожидании, когда данные из оперативной памяти будут доставлены аппаратными средствами в более быстродействующую память устройств управления.

Были испробованы два подхода к решению этой проблемы. В первом случае было разработано программное обеспечение, позволяющее эффективнее использовать аппаратную часть компьютера. Операторы, подобные оператору and, обсуждавшемуся в разделе 11.2.1, позволяют программисту создавать параллельно выполняющиеся программы. Это дает возможность более эффективной работы аппаратной части, поскольку позволяет выполнять другую программу в то время, когда выполнение первой программы заблокировано из-за ожидания данных.

Альтернативный подход заключается в разработке более эффективной аппаратуры. Выпуск центральных процессоров с большей по размерам управляющей памятью уменьшает количество необходимых обращений к оперативной памяти за требуемыми данными. Использование кэш-памяти между оперативной и управляющей памятью позволяет значительно ускорить работу последней. Тем не менее, как бы ни возрастало быстродействие компьютеров, постоянно создаются приложения, которые требуют все больших компьютерных ресурсов.

Помимо того, меняется сама природа компьютерных вычислений. Теперь компьютеры уже не являются изолированными вычислительными машинами, существующими сами по себе. Миллионы настольных компьютеров представляют собой мощные рабочие станции. Текст данной книги был набран на персональном компьютере, подключенном к международной сети Интернет. Концепция клиент-серверных вычислений, когда вычислительные задачи распределены среди мно-

жества компьютеров, меняет в корне природу программного обеспечения. Хотя, как подчеркивается в разделе 4.2, мы заинтересованы в том, чтобы проверить корректность программы и убедиться в том, что она завершится; для многих приложений, например систем управления телефонной сетью, предварительного заказа авиабилетов или бронирования мест в гостинице, понятие «завершение приложения» вообще отсутствует, так как подобные программы *никогда* не должны завершаться. Завершение их работы могло бы повлечь за собой значительные потери в прибыли для пользователей таких приложений.

Чтобы разобраться в этих вопросах, мы кратко обсудим некоторые современные тенденции в компьютерных вычислениях, а также приведем наши предположения о том, как эти тенденции отразятся на идеологии языков программирования. Мы разделяем тенденции развития аппаратной части компьютера и программного обеспечения.

Как отмечалось в разделе 2.1.1, компьютер состоит из центрального процессора (ЦП), выполняющего команды, небольшой по размерам, но быстродействующей управляющей памяти и большей по размеру, но и более медленной оперативной памяти. Из-за различия скоростей доступа к памяти ЦП и оперативной памяти центральному процессору часто приходится ждать получения данных — это так называемая *проблема узкого места* фон Неймана. В главе 2 говорилось, что кэш-память — одно из аппаратных решений этой проблемы, которое полностью незаметно для программиста. Аппаратная часть компьютера автоматически управляет кэш-памятью, так что ее добавление в компьютерную систему весьма незначительно влияет на разработку языка программирования.

Это утверждение, впрочем, не является абсолютно точным. Управление кэш-памятью может повысить производительность работы компьютера. Если, например, цикл в программе занимает 32 байт и каждый сегмент кэш-памяти также имеет длину 32 байт, можно значительно увеличить эффективность работы программы, разместив начало цикла на границе сегмента кэш-памяти, что позволит впоследствии весь цикл хранить в одном сегменте кэш-памяти вместо того, чтобы использовать для него два сегмента кэш-памяти. Таким образом можно высвободить дополнительный сегмент кэш-памяти для других данных. Хотя при этом кэш-память получает дополнительный сегмент и может работать несколько быстрее, этот выигрыш в скорости будет незначительным по сравнению с гораздо большим выигрышем, обусловленным применением самой кэш-памяти.

К увеличению общей производительности аппаратной части компьютера существует два основных подхода:

1) повысить производительность отдельного ЦП;
2) разработать альтернативные архитектуры, которые позволят увеличить общую производительность.

В следующих двух разделах мы обсудим каждый из этих подходов.

11.3.1. Конструирование процессоров

Очевидный метод увеличения эффективности работы компьютера заключается в том, чтобы увеличить скорость ЦП. В этом разделе мы сначала обсудим компьютер с центральным процессором, имеющим стандартную архитектуру, который

часто называется компьютером с полным набором команд (complex instruction set computer, CISC). Затем мы рассмотрим компьютер с сокращенным набором команд (reduced instruction set computer, RISC) как метод улучшения производительности системы.

CISC-компьютеры. Один из ранних подходов к увеличению эффективности работы компьютеров связан с минимизацией перемещений данных между оперативной памятью и ЦП. Если бы команды стали более мощными, потребовалось бы меньшее их количество, даже если при этом понадобилось бы большее время на выполнение каждой такой команды. Такой подход привел к появлению CISC-архитектуры. Хотя в первых машинах обычно имелся только ограниченный набор команд (например, перемещение данных в оперативную память или из нее, простые арифметические операции с данными в управляющей памяти, переход к адресу новой команды), в современных машинах может быть представлено до сотни доступных операций. Каждая операция часто разбивается на подполя:

```
[операция][данные источника][данные назначения]
```

Источником для команды может служить некоторая область основной или управляющей памяти, исполнительный адрес в основной памяти или некоторое значение-константа. То же самое (за исключением значения-константы) относится и к назначению. Операция может быть операцией, работающей с байтами или словами, арифметической (например, + или ×), логической (например, ∧ или ∨) или условной операцией (например, > или ≤) или операцией перехода (например, goto). Сложные операции, такие как вычисление индекса цикла do, доступ к элементам массива, преобразование целочисленного формата в вещественный, часто представляются в виде одной встроенной операции благодаря их широкому применению в большинстве языков программирования.

CISC-архитектура налагает на разработчика компилятора бремя обеспечения эффективного использования новых команд. Например, в мире персональных компьютеров процессор Intel 80286 использовал 16-битные данные, и операционная система MS-DOS была создана для работы на машине именно с таким процессором. Эта операционная система также функционирует и на компьютерах с более быстрыми процессорами i486 и Pentium, в которых используются 32-битные команды. Но программы, разработанные для старого процессора 80286, не использовали эффективно новую аппаратную часть, и производители компиляторов были вынуждены модифицировать свои компиляторы, чтобы они генерировали команды, которые эффективно использовали более мощные команды более быстродействующих центральных процессоров.

RISC-компьютеры. Добавление все более и более сложных команд означает, что требуется все большее количество циклов центрального процессора для выполнения каждой команды. Хотя скорость работы компьютеров постепенно увеличивалась сперва до 100 МГц, а затем до 200 МГц, 600 МГц и более, количество циклов, необходимых для выполнения каждой инструкции, также увеличивается от 1 до 2, 3, 4 и т. д. Таким образом, хотя компьютеры становятся все более быстродействующими, сложность используемых центральных процессоров также возрастает — например, в современных микропроцессорах один чип состоит из более чем 3 млн компонентов. Такая сложность часто приводит к аппаратным ошибкам в архитектуре чипа.

Для того чтобы остановить неограниченное увеличение быстродействия и возрастание сложности чипа в 70-е гг. была предложена альтернативная RISC-архитектура. Идея заключалась в том, чтобы уменьшить насколько это возможно сложность ЦП и извлечь всю выгоду из скорости выполнения машинного цикла. Так, машина с тактовой частотой процессора 400 МГц должна выполнять 400 млн команд в секунду. Хотя такой машине, возможно, потребуется выполнить большее количество команд, чем соответствующей программе на CISC-компьютере, общее убеждение заключалось в том, что увеличение скорости машины даст больший эффект, чем использование дополнительных инструкций.

RISC-процессор соединяет несколько принципов проектирования, которые способствуют быстрому выполнению.

- *Выполнение за один цикл.* Каждая команда должна выполняться за один цикл (при условии конвейерной обработки).
- *Конвейерная архитектура.* Выполнение команды требует несколько этапов — например, следующие:
  1) извлечь команду из оперативной памяти;
  2) декодировать поле операции, данные источника и данные назначения;
  3) получить данные источника для операции;
  4) выполнить операцию.

Конвейерная архитектура выполняет эту последовательность в несколько этапов с помощью последовательных команд на каждой стадии. Так, пока выполняется операция `команда 1`, `команда 2` получает данные источника, `команда 3` декодируется, а `команда 4` извлекается из памяти. Для выполнения каждой отдельной команды необходимо четыре машинных цикла, но каждый цикл завершает выполнение очередной команды, выполняя, таким образом, команды со скоростью полного цикла ЦП, если только удается всегда держать четыре команды в конвейере. Выполнение этого условия, однако, является весьма сложным делом и, как будет показано далее, ставит перед разработчиком языка непростые задачи.

- *Большая управляющая память.* Чтобы избежать частых обращений к оперативной памяти, используется большой набор (часто 100 и более) регистров. Это увеличивает количество элементов данных в управляющей памяти, позволяя реже обращаться за данными в оперативную память.
- *Быстрое инициирование процедур.* Размещение записей активации в оперативной памяти (см. главу 9) часто требует больших затрат времени. При использовании большого количества управляющих регистров в RISC-процессоре стек записей активации подпрограмм во время выполнения часто размещается в управляющей памяти. Вызов подпрограммы и возврат из нее часто реализуются в виде соответствующих команд RISC-процессора.

Такая идеология процессора должна учитываться разработчиками трансляторов. Например, в стандартном CISC-компьютере оператор `E = A + B + C + D` в постфиксной записи будет иметь вид `EAB + C + D + =` и выполняться следующим образом:

1) добавить A к B, получить сумму;

2) к сумме добавить C;

3) к сумме добавить D;

4) сохранить сумму в E.

Но в конвейерной архитектуре команда 2 (к сумме добавить C) не может получить необходимое ей значение суммы (результат сложения A и B — этап 3 в нашем примере с конвейером), пока предыдущая команда (добавить A к B) не сохранит свой результат (этап 4 в конвейере). В итоге процессор вынужден ждать один цикл при выполнении команды 2, пока команда 1 не завершит свое выполнение. Более «сообразительный» транслятор предложил бы другую постфиксную запись, EAB + CD + + =, которая позволила бы вычислять A + B одновременно с C + D и, таким образом, избежать задержки. Еще более «сообразительный» транслятор мог бы чередовать выполнение отдельных операций двух независимых операторов, например:

```
E=A+B+C+D:
J=F+G+H+I
```

со следующей постфиксной записью

```
AB + FG + CD + HI + (1)(3) + (2)(4) + E(5) = F(6) =,
```

где номера операндов означают *номера операций* в этом постфиксном выражении. В этом случае каждый оператор выполняется без взаимного влияния со стороны другого и процессор работает с максимальной скоростью, возможной на конвейере. Но при этом транслятору языка приходится обеспечивать отсутствие взаимного влияния операторов. Например, если бы второй оператор был J := E + G + H + I, то операция EG+ не могла бы выполниться, пока не завершилось бы выполнение операции присваивания переменной E из первого оператора. Что еще хуже, для максимального упрощения структуры процессора вся ответственность за подобные решения часто возлагается на транслятор языка; аппаратная часть компьютера просто выбирает соответствующие, возможно, не корректные значения и продолжает свой цикл выполнения.

Команды передачи управления налагают дополнительные ограничения. Например, для быстрого выполнения оператора if

```
if expr then stmt1 else stmt2
```

последовательность выполнения на большинстве RISC-компьютеров будет следующей:

- вычислить выражение expr;
- поместить $stmt_1$ в конвейер;
- если выражение expr ложно, то очистить конвейер и поместить в него $stmt_2$.

Предполагая, что выражение будет истинно, RISC-процессор может начать выполнение ветви then без остановки процессора на несколько циклов в ожидании того, что конвейер «поймает» необходимую последовательность операций. Но если выражение ложно, важно то, что никакие побочные вычисления, относящиеся к ветви then, не будут сохранены. Разработка программ, в которых большинство ветвей истинно, могло бы ускорить выполнение на машинах с RISC-архитектурой.

11.3.2. Конструирование систем

Увеличение скорости работы аппаратной части компьютера обычно достигается одним из двух способов. Первый способ заключается в том, что на RISC-компьютерах с конвейерной архитектурой несколько команд, обрабатывающих одни и те же данные, выполняются одновременно. Альтернативный подход представляет компьютер со множеством потоков команд и множеством потоков данных[1] (Multiple Instruction Multiple Data, MIMD), в котором различные инструкции, выполняясь одновременно, обрабатывают различные данные.

Мультипроцессоры не являются новейшей разработкой. Два и более ЦП, работающих под управлением одной операционной системы, известны вот уже около двадцати лет. В простейшем случае операционная система просто назначает каждой пользовательской программе свой процессор, и взаимное влияние этих программ практически отсутствует. Суммарная производительность такой системы возрастает за счет наличия двух процессоров, но на скорость выполнения отдельной программы это практически не влияет.

На рис. 11.5 представлены три подхода к архитектуре системы с несколькими ЦП. На рис. 11.5, *а* представлен обычный подход к использованию нескольких процессоров в одной системе. Несколько процессоров взаимодействуют с несколькими модулями памяти посредством одного тракта данных, называемого *магистральной шиной*. Эта архитектура является простейшей из всех рассматриваемых нами, но у нее есть относительно серьезный недостаток, заключающийся в том, что может возникнуть взаимное влияние доступа к данным различных процессоров. По этой причине общая производительность системы, состоящей из p процессоров с частотой k мегагерц каждый, будет значительно меньше, чем $p \times k$.

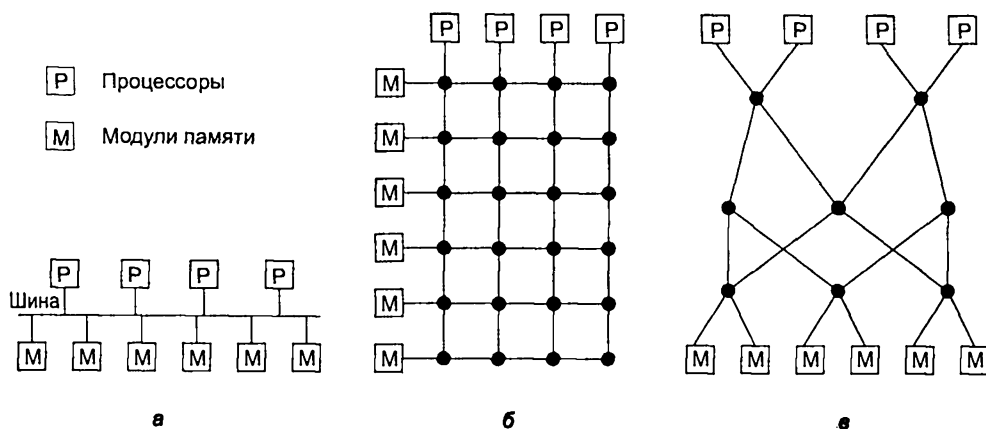


Рис. 11.5. Системные архитектуры для мультипроцессоров: а — одна шина; *б* — матричный переключатель; *в* — омега-сеть

[1] В отечественной литературе по вычислительной технике такие компьютеры еще называют компьютерами с векторно-конвейерной архитектурой. — *Примеч. науч. ред.*

Вторая модель — это матричный переключатель, схематически представленный на рис. 11.5, *б*. В этой модели каждый процессор может быть по отдельности связан с отдельным модулем памяти. Хотя здесь мы можем достичь производительности $p \times k$ мегагерц, но если у нас имеется m модулей памяти, то потребуется $p \times m$ переключателей. Это значительно увеличивает стоимость подобных систем.

Промежуточным, компромиссным решением является так называемая архитектура *омега-сети* (см. рис. 11.5, *в*). Здесь имеется набор переключателей, который позволяет связать по определенному маршруту каждый процессор с каждым модулем памяти. По сравнению с архитектурой матричных переключателей несколько уменьшается скорость из-за необходимости установления сетевого маршрута к данным, но по сравнению с архитектурой магистральной шины имеется выигрыш в скорости, так как будет возникать несколько меньше случаев взаимного влияния данных. Стоимость такой архитектуры меньше по сравнению с архитектурой матричных переключателей, так как требуется меньшее число переключателей, но выше по сравнению с архитектурой магистральной шины.

С помощью каждого из представленных архитектурных решений операционная система может просто назначить выполнение различных задач на различных процессорах. В таком случае аппаратная часть никак не влияет на разработку языка программирования, и каждый процессор просто выполняет последовательную программу. Но если операционная система выделяет несколько процессоров для решения одной задачи, то в таком случае разработка языка играет решающую роль и в ней следует обратить внимание на механизмы реализации синхронизации задач, описанные в разделе 11.2.5.

Компьютеры с массовым параллелизмом. С использованием мультипроцессорной архитектуры мы можем иметь сотни и даже тысячи взаимосвязанных процессоров. Такие машины мы называем *компьютерами с массовым параллелизмом*. И здесь возникает задача: как разделить программу на достаточно небольшие фрагменты, которые каждый процессор мог бы выполнять независимо без того, чтобы потребности в данных одного процессора постоянно взаимодействовали с потребностями в данных другого процессора.

Мы можем посмотреть на проблему параллельной организации вычислений с точки зрения *низкоуровнего* или *высокоуровнего* подхода. Высокоуровневые подходы зависят от конструкций языка, которые использованы программистом для организации параллельного выполнения. Оператор and из раздела 11.2.1 является одним из методов, который может использовать программист, чтобы указать транслятору планировать выполнение каждого отдельного оператора на отдельном процессоре. К сожалению, программисты редко способны вручную организовать параллельное выполнение программ эффективным образом, так что на практике такие подходы хорошо не работают.

При низкоуровневом подходе элементарные конструкции языка, вычисление которых можно осуществить параллельно, определяются транслятором. Наиболее значимые примеры можно найти при обработке массивов. Во многих случаях (например, при моделировании потоков жидкости, потоков воздуха, обтекающих крыло самолета, или при прогнозировании погоды) точность решения зависит от числа пробных точек, используемых для исследования физических явлений. Нередко встречаются матрицы, состоящие из миллионов точек, и их перемножение

требует миллионов или миллиардов операций. Поскольку при перемножении матриц производится умножение i-й строки матрицы A на j-й столбец матрицы B для всех i и j, то кэш-память фактически не оказывает влияния на производительность, так как данные приходится извлекать практически из любой области оперативной памяти. Вместо этого при низкоуровневом подходе к параллельному вычислению матрица разбивается на небольшие сегменты, каждый из которых обрабатывается отдельным процессором. Часто такие матрицы являются *разреженными*, то есть большинство их элементов представлено нулями. В таком случае многие сегменты матриц также являются нулевыми и их можно не учитывать при умножении. Языковые конструкции типа сечений важны при создании алгоритмов, использующих преимущества архитектур низкоуровневого параллельного вычисления.

11.4. Архитектура программного обеспечения

Большинство языков программирования основано на дискретной модели выполнения. Это означает, что программа начинает выполнение, а затем в определенном порядке она:

1) считывает соответствующие данные из локальной файловой системы;
2) генерирует некоторые результаты;
3) записывает данные в локальную файловую систему.

Это описание ясно показывает существование двух классов данных: *временных* (transient) данных, время жизни которых ограничено временем выполнения программы, и *сохраняемых*, или *хранимых* (persistent), данных, время жизни которых превосходит время выполнения программы. Традиционно временные данные — это объекты, определяемые используемым языком программирования, и большинство рассмотренных ранее объектов принадлежит именно к этому классу. Сохраняемые данные в основном относятся к области систем баз данных или файловых систем. При выполнении программы сохраняемые данные считываются программой во временные данные, а затем временные данные перезаписываются обратно в постоянные хранилища сохраняемых данных.

11.4.1. Сохраняемые данные и системы транзакций

Для большинства приложений разделение данных на временные и сохраняемые объекты довольно удобно, и оно отлично служило нам в течение многих лет. Тем не менее существуют приложения, в которых эта модель оказывается совершенно неэффективной. Рассмотрим систему предварительных заказов (например, билетов на самолет или номеров в гостинице). Желательно, чтобы такая система работала непрерывно, так как заказ может появиться в любое время. Но в таком случае весьма неэффективными окажутся считывание хранимых данных, описывающих состояние заказов в целом, изменение этого состояния с учетом вновь поступив-

шей информации и последующая перезапись заказов обратно в места их постоянного хранения.

Систему предварительных заказов можно было бы переделать таким образом, чтобы вся база данных была представлена в программе в виде обычного массива, а сама программа в бесконечном цикле ожидала бы новых запросов. Это позволило бы избежать необходимости повторного чтения данных каждый раз, когда требуется обновление базы данных заказов. Проблема при таком решении возникнет, если произойдет отказ в работе программы (например, из-за программной ошибки или снижения напряжения в сети электропитания), — все временные данные будут потеряны. Мы лишились бы содержимого нашей системы предварительных заказов.

Эта проблема решается с помощью языка программирования, *поддерживающего сохраняемые данные*. В данном случае различие между сохраняемыми и временными данными отсутствует. Все изменения, вносимые программой во временные переменные, немедленно отражаются в изменении базы данных сохраняемых значений, и при этом нет необходимости в начале выполнения программы считывать данные из постоянного места хранения во временное. Приведенные две концепции являются совершенно идентичными, приводящими к надежности системы хранения данных, если предполагаются отказы системы во время ее функционирования. Под языком программирования с сохраняемыми данными мы понимаем такой язык, в котором явным образом не указывается перемещение данных (например, между временным и постоянным местом хранения).

В языке Smalltalk (раздел 7.2.5) присутствуют некоторые аспекты сохраняемости. Пользовательская среда классов и методов остается неизменной от одного выполнения к следующему. Тем не менее локальные данные какого-либо метода утрачиваются после завершения его работы. Существует несколько языков, поддерживающих концепцию сохраняемых данных (например, χ [94]), которые действительно сохраняют целостность данных в случае отказа программы.

Разработка языка программирования с сохраняемыми данными требует использования тех же самых концепций, которые применяются в разработках традиционных языков типа C или Pascal. Однако поддержка сохраняемых данных накладывает несколько дополнительных проектных ограничений на разработку языка [94].

1. *Необходим механизм для обозначения того, что данный объект является сохраняемым.* Обычный подход заключается в предположении, что все данные являются временными, и следует указать, какие из них должны быть сохраняемыми.

2. *Необходим механизм для адресации сохраняемых объектов.* Часто используется файловая структура операционной системы, хотя это и не обязательно. В языке может быть разработана своя модель хранения подобных объектов. Однако необходим механизм, позволяющий двум различным программам, разработанным с помощью этого языка, получать доступ к одному и тому же сохраняемому объекту.

3. *Необходимо синхронизировать одновременный доступ к индивидуальному сохраняемому объекту.* Для этого существуют хорошо разработанные методы, например использование семафоров (см. раздел 11.2.5).

4. *Необходимо проверять совместимость типов сохраняемых объектов.* Поскольку один и тот же сохраняемый объект может использоваться в различных программах, то эквивалентность объектов на основе эквивалентности имен использовать довольно трудно. По этой причине структурная эквивалентность является предпочтительным методом в языках с сохраняемыми данными.

11.4.2. Сети и клиент-серверные вычисления

Все приведенные рассуждения наводят на мысль, что программы обычно пишутся для доступа к локальным хранилищам данных и получения некоторых результатов, после чего они должны завершаться. Компьютер при этом рассматривается как машина, используемая для решения поставленной задачи. По мере того как машины становились более быстродействующими и дорогими, разрабатывались мультипрограммные системы и к машинам добавлялись терминалы, чтобы позволить большему количеству пользователей одновременно получить доступ к компьютеру.

Ситуация начала меняться в начале 80-х гг. в связи с появлением мини-компьютеров и позже — микропроцессоров (то есть персональных компьютеров). По мере того как машины уменьшались в размерах и становились территориально рассредоточенными, возник интерес к развитию технологий, обеспечивающих возможность коммуникации между этими распределенными машинами. Начавшись с сети ARPANET (Advanced Research Project Agency Network) в 70-х гг., коммуникационные технологии на основе высокоскоростных линий связи продолжали развиваться с распространением Интернета в 90-х гг. прошлого столетия. В настоящее время практически все компьютеры имеют возможность связываться с другими компьютерами с помощью высокоскоростных линий связи. Такие протоколы, как X.25 и TCP/IP обеспечивают надежность передачи информации между двумя рассредоточенными компьютерами. Мы называем процессоры, связанные коммуникационными линиями связи, компьютерной *сетью.*

В разделе 11.3 описаны системы *с непосредственными связями* (сильно связанные системы). В них выполнение нескольких программ осуществляется параллельно с использованием мультипроцессорных архитектур и такими аппаратными технологиями, как, например, конвейерная обработка. Использование линий связи позволяет разрабатывать *слабосвязанные* системы. В таких системах каждый процессор имеет свою собственную оперативную память и собственное дисковое запоминающее устройство, но взаимодействует с другими процессорами для передачи информации. Системы, разработанные на основе таких архитектур, называются *распределенными системами.*

Распределенные системы могут быть *централизованными,* когда какой-либо один процессор осуществляет планирование и сообщает остальным машинам, какие задачи им следует выполнять, или *равноправными,* когда все машины равноправны и участвуют в процессе планирования. В централизованной системе один процессор будет сообщать другому процессору, какую задачу тот должен выполнить. По ее завершении второй процессор сигнализирует, что он готов к выполнению следующей задачи. В равноправной системе один процессор может послать

широковещательное сообщение всем процессорам о том, что требуется выполнить некоторую задачу, и какой-нибудь свободный в данный момент процессор может ответить сообщением: «Я ее выполню». В отличие от предыдущего способа каждая машина в сети будет запускать задачу на некоторой другой машине той же сети. При таком способе вся сеть постепенно загружается работой.

Развитие персональных компьютеров изменило эту модель. В начале 80-х гг. использование терминалов являлось типичным методом взаимодействия с большой универсальной машиной. Когда персональные компьютеры (ПК) стали появляться на рабочих столах, для них были разработаны программы, имитирующие работу терминала. Таким образом, ПК могли работать локально как отдельные компьютеры, а также могли служить терминалом для больших универсальных машин.

По мере того как ПК становились более быстродействующими, выяснялось, что они могли бы решать вычислительные задачи, которые до сих пор выполнялись только на больших универсальных машинах. Для этого данные из большой машины можно было бы передавать на ПК и обрабатывать локально. Например, финансовая база данных большой компаниыи могла храниться на большой универсальной машине, а программа табличных вычислений для получения локальных статистических данных могла выполняться на ПК. Программа, работающая на большом компьютере, стала называться *сервер*, тогда как программа, выполняющаяся на ПК, стала называться *клиент*. Клиент-серверные вычисления предполагают разделение задач на сегменты — части, требующие централизованного управления, например поиск в базе данных, и части, требующие локальных вычислений, например табличные вычисления.

По мере того как мощность микропроцессоров увеличивалась и дальше, многие приложения, которые выполнялись на универсальных машинах, начинали легко обрабатываться и на рабочих станциях. Использование больших универсальных машин существенно сократилось, но клиент-серверная модель вычислений все еще оставалась жизнеспособной. В настоящее время существует множество приложений, разработанных с использованием этого подхода. Например, централизованное хранилище данных осуществляет и извлекает данные как программа-сервер, в то время как программа-клиент выполняется на локальной рабочей станции, осуществляя вычисления, электронную обработку текстов или обрабатывая данные каким-либо иным способом.

Основное влияние этой концепции на разработку языков программирования проявилось в отказе от хранения глобальных данных для программы. И сервер, и клиент имеют лишь ограниченный доступ к информационному содержимому программы. Для эффективного вычисления результата программы необходимо разбить на отдельные части. Некоторые методы решения этих проблем уже обсуждались в нашей книге. Задачи в языке Ada представляют одну из форм разделения программ на независимые части. Другим методом является вызов удаленных процедур[1]

---

[1] Вызов удаленных процедур был первым способом реализации распределенных приложений, позволявший передавать в качестве параметров процедур и возвращать результаты в виде основных элементарных типов данных. В настоящее время в связи с широким распространением объектно-ориентированного подхода разработаны специальные технологии вызова методов удаленных объектов и возвращения результатов в виде объектов: DCOM, RMI, CORBA. — *Примеч. науч. ред.*

(Remote Procedure Call, RPC). Синтаксически вызов удаленной процедуры выглядит так же, как обычный вызов процедуры, за исключением того, что операционная система будет осуществлять вызов программы, выполняемой, возможно, другим процессором. Вызовы удаленных процедур связаны с передачей сообщений и часто реализуются с помощью методики сообщений *отправить-получить* (send-receive), при которой клиент посылает сообщение о вызове удаленной процедуры на сервер, ожидающий *получения* сообщения и затем отвечающий клиенту аналогичным способом.

Широкое распространение технологии WWW вызвало интерес к разновидности клиент-серверной архитектуры вычислений с промежуточным звеном[1]. Промежуточное звено — это программный модуль, который использует закодированную информацию о множествах данных и создает информацию, необходимую для класса приложения более высокого уровня. При такой организации пользователь взаимодействует с программным обеспечением клиента (например, web-браузером), который, в свою очередь, взаимодействует с сервером. У сервера имеется доступ к многочисленным ресурсам данных, и он извлекает информацию из некоторых ресурсов, чтобы создать новую информацию для пользователя. Более подробно мы обсудим эти вопросы в разделе 12.2.

Вообще говоря, клиент-серверное программирование не требует внесения больших изменений в используемый язык программирования, если в него добавлен какой-либо механизм сообщений, например вызов удаленных процедур или какой-либо другой. Но структура самой программы становится более сложной. Программа должна быть разделена на фрагменты таким образом, чтобы минимизировать передачу сообщений по сети. Например, можно перемножать матрицы в программе-клиенте, каждый раз запрашивая с сервера очередную строку или столбец, но результирующий трафик сообщений может перегрузить сеть и сделать систему полностью непригодной к работе.

11.5. Рекомендуемая литература

Параллельные системы обсуждаются в [26, 121]. Устройство RISC-машин рассматривается в статьях [24, 60 и 102], а в [111] описано несколько примеров RISC-архитектуры. В статье [73] описана разработка параллельных алгоритмов для MIMD-машин. Общий обзор вариантов организации параллельной архитектуры дается в [38]. Компьютеры с массовым параллелизмом обсуждаются в [42, 112]. Языки, поддерживающие сохраняемые данные, представлены в [13].

11.6. Задачи и упражнения

1. Возможно ли реализовать операции `wait` и `signal` семафора с помощью сообщений `send` и `receive`? Напишите две процедуры `signal(P)` и `wait(P)`,

[1] Архитектура системы клиент-сервер с промежуточным звеном называется также трехслойной. — *Примеч. науч. ред.*

которые взаимодействуют с помощью сообщений, но семантически имеют тот же смысл, что и операции семафора wait и signal. Какие проблемы с атомарностью возникают при таком решении? Операции wait и signal могут использоваться множеством задач, применяющих один и тот же семафор. Как можно обобщить сообщения send и receive, чтобы они могли использоваться более чем одной посылающей и одной принимающей задачей?

2. Одной из наиболее трудных задач в параллельных приложениях является получение ответа на вопрос «который час?». Синхронизация часов в сети компьютеров может оказаться сложным делом, так как мы хотим, чтобы отсчитываемое на разных компьютерах сети время было бы абсолютно одинаковым, но сама передача точного времени на все компьютеры требует некоторого конечного промежутка времени.

   а) Предложите метод сообщения значения времени всем процессорам в мультипроцессорной системе, так чтобы в результате на всех машинах время было в точности одинаковым.

   б) Если в некоторой системе имеются отдельные функции для определения текущих значений времени и даты, к каким проблемам это может привести? (Рассмотрите проблему функциональной атомарности при определении времени близ полуночи. По этой причине в большинстве систем в настоящее время используется одна функция TimeAndDate для определения текущих времени и даты.)

3. Передача параметров по ссылке и по значению-результату обычно дает один и тот же эффект (с точки зрения программиста). Но если подпрограмма завершается в результате возникновения исключительной ситуации и это исключение передается обработчику, расположенному вне данной подпрограммы (или оператором goto осуществляется переход к помеченному нелокальному оператору), два метода передачи параметров могут приводить к различным эффектам. Предположим, что в подпрограмме Sub1 параметр Y имеет значение 5, затем этот параметр передается в подпрограмму Sub2, в ней этому параметру присваивается значение 7 и затем генерируется исключение, которое передается обработчику из подпрограммы Sub1. Поясните, какое значение переменная Y может иметь в подпрограмме Sub1 после обработки исключения, если:

   а) передача происходила по ссылке;

   б) передача происходила по значению-результату.

4. Рассмотрим систему из нескольких процессоров, каждый из которых имеет кэш-память. Проблема когерентности кэш-памяти возникает, когда один процессор обновляет данные в своей кэш-памяти, в то время как другому процессору требуется получить доступ к этим данным до того, как первый процессор успеет перезаписать содержимое кэш-памяти в основную память. Так как кэш-память связана с ЦП, а не с оперативной памятью, обсудите способы организации аппаратной части, которая позволила бы решить эту проблему.

5. В некоторых RISC-процессорах выполнение оператора присваивания

```
while некоторое условие do
    - Здесь помещаются некоторые операторы
    C=A+B
    end
```

происходит следующим образом:

а) в управляющую память загружается A;

б) в управляющую память загружается B;

в) два значения складываются (то есть вычисляется A + B);

г) переход на оператор while;

д) содержимое управляющей памяти сохраняется в C.

Объясните, как запись содержимого управляющей памяти в переменную C может происходить *после* перехода к следующему шагу оператора цикла while и все же быть правильной.

6. *Вызов удаленной процедуры* имеет синтаксис вызова функции, например:

```
call NewFunction (p1, p2, p3)
```

Отличие заключается в том, что вызываемая процедура может выполняться на каком-либо другом компьютере, подключенном к сети. Вызванной процедуре передаются значения фактических параметров, а полученный результат пересылается обратно в вызывающую подпрограмму. Объясните, как можно реализовать вызов удаленной процедуры с использованием сообщений в качестве механизма синхронизации. Как бы вы подошли к решению этой задачи в случае, если бы различные параметры могли передаваться:

а) по значению;

б) по ссылке;

в) по результату?

Глава 12. Сетевое программирование

Многие думают, что компьютеры — это просто большие *калькуляторы* или сверхбыстродействующие *вычислительные комплексы,* — и это лишь потому, что вначале они использовались именно для вычислений: расчетов баллистических таблиц, решения дифференциальных уравнений при прогнозировании погоды или для составления платежных ведомостей. Однако природа компьютерных вычислений изменяется, так как сегодня мы используем компьютеры для пилотирования самолетов, управления автомобилями, обработки документов, игр и даже общения с нами на человеческом языке. Разработка языков программирования должна быть в согласии с тенденциями к не числовым приложениям. Влияние некоторых новых тенденций уже рассматривалось в данной книге. На примере языков Prolog, LISP и Java можно видеть, насколько различными могут быть данные, обрабатываемые программой. В этой главе мы проследим за эволюцией текстовой информации. В главе 11 представлены некоторые начальные понятия о распределенной обработке данных и клиент-серверном программировании, но появление настольных издательских систем, бурный рост Интернета и WWW привели к значительному усилению интереса к языкам для работы с текстами. В этой главе мы рассмотрим Postscript как язык для описания документов. Затем мы обсудим WWW и различные языки программирования, которые используются для управления информационными потоками.

Модели трансляторов. Приступая к обсуждению обработки текстов, мы должны расширить наши представления о компиляторе. В главе 2 мы рассматривали интерпретаторы, виртуальные машины и времена связывания при исследовании вопроса о взаимодействии трансляторов с данными программы. Для понимания приложений обработки текстов нам придется посмотреть на трансляторы с несколько другой точки зрения. Прежде всего мы можем разбить все трансляторы на три большие группы в соответствии с формой производимых ими выходных данных.

1. *Интерпретация.* Выводом транслятора является решение поставленной перед ним задачи. Например, в ответ на вопрос: «В чем смысл жизни, времени и всего сущего?»[1] — транслятор просто напишет число 42[2]. Для кода

```
sum=0;I=0;
while I<10 do
    { I=I+1; sum=sum+I; }
```

---

[1] Из великолепной книги Дугласа Адамса (Douglas Adams) «Hitchhiker's Guide to the Galaxy» [6].

[2] Не спрашивайте нас, почему 42, — спросите мистера Адамса.

транслятор просто произвел бы код `sum = 55`. В данном примере транслятор действует как интерпретатор и выполняет запрос на вывод желаемого ответа.

2. *Компиляция.* В этом случае транслятор производит алгоритм (то есть программу), которая и будет вычислять ответ. Это соответствует типичному компилятору. Компилятор С создает программу для аппаратной вычислительной машины, которая может выполнить ее и произвести желаемый ответ, например, для приведенной выше программы на С. Существует несколько распространенных типов машинной архитектуры (например, Intel Pentium, Apple PowerPC, различные RISC-процессоры). Компилятор будет создавать уникальный для каждой конкретной машинной архитектуры набор команд, который и будет использоваться для вычислений.
3. *Семантическое описание.* В этом случае транслятор производит описание вывода. Например, вместо простой записи 42 вывод на вопрос «В чем смысл жизни, времени и всего сущего?» мог быть дан в виде `<integer><binary value> 101010 <end>`. Заметим, что это более общий ответ, чем простая запись 42, которая предполагает, что пользователь знает контекст этого ответа, — числа записываются в десятичной системе счисления. Например, на планете Фраммис, жители которой используют восьмеричную систему счисления, число 42 в двоичной записи будет выглядеть как 100010, что соответствует числу 34 в десятичной записи, а это совершенно другое значение. Однако семантическое описание включает семантическую информацию, необходимую для интерпретации ответа в произвольной системе счисления и печати результата с использованием любого набора символов, — это не обязательно должны быть привычные нам арабские цифры.

Для численных расчетов почти во всем мире используются арабские цифры. Но такого единообразия не может быть в случае обрабатывающих тексты приложений. Те, для кого родными являются такие языки, как иврит, японский, арабский, корейский и т. п. должны иметь возможность использовать свой алфавит. В языках программирования, предназначенных для обработки текста, наряду с получением ответа наиболее важное значение имеет возможность отслеживания среды, где это приложение будет использоваться. Таким образом, для текстовых приложений три вида ранее описанных трансляторов приобретают следующий смысл.

1. *Интерпретация.* Этот вид трансляторов будет представлять определенный формат вывода. Для текста таковым могло бы быть множество битов, представляющее страницу, которую требуется отобразить. При ширине в 1000 бит и высоте в 1200 бит потребовалось бы 1 200 000 бит для описания черно-белой страницы или приблизительно 150 Кбайт данных. Хотя технологии сжатия данных могут значительно уменьшить этот объем, все же представленный метод создания и хранения текстовой информации является весьма негибким. Как правило, он практически не используется для таких приложений.
2. *Компиляция.* В этой модели транслятор преобразует документ в выполняемую программу, результатом работы которой является создание его изображения; то есть программу, написанную для виртуальной машины, выполняющей набор команд для создания этого изображения. В языке Postscript

используется именно такой процесс. Виртуальная машина Postscript принимает команды Postscript, результат выполнения которых можно сравнить с рисованием на чистом листе бумаги. Когда «лист» закончен, команда печати отображает на экране монитора или печатает на принтере полученную страницу. Трансляторы Postscript производят программу, которая создает окончательный документ в его форматированном описании для выполнения на виртуальной машине Postscript. В разделе 12.1 дается более полное описание языка Postscript.

3. *Семантическое описание.* В этом случае мы описываем атрибуты окончательного документа, но не детали того, как он в действительности выглядит. Например, компания Microsoft для сохранения документов использует формат RTF (Rich Text Format) с помощью приложений своего текстового процессора Word. Для чтения и записи документов в формате RTF можно написать и другие текстовые процессоры. В формате RTF используется специальный набор команд, описывающих каждый атрибут документа, подобно тому как число 42 было определено с атрибутами `integer` (целый) и `binary value` (двоичное значение) в приведенном выше примере. Для технологии WWW стандартным способом описания текстовых данных стал язык гипертекстовой разметки (HyperText Markup Language, HTML). В разделе 12.2 мы опишем HTML, а также рассмотрим язык Java как расширение возможностей web-приложений по обработке документов.

12.1. Настольные издательские системы

Описываемые в данной книге языки программирования основаны на традиционном подходе к трансляции — текст исходной программы компилируется в выполняемый на виртуальной машине код. Под управлением этой машины данные читаются и обрабатываются, а затем печатаются полученные результаты. Неотъемлемой частью такой модели программирования является то, что программа сначала *реализуется*, а затем уже *пишется* документация для работы с этой программой.

Но документация является всего лишь другой формой данных, и при написании документов используются концепции, близкие к концепциям написания программ. Например, структура данной книги определена в терминах глав и разделов внутри глав. По мере написания текста определялись ссылки на конкретные его фрагменты и другие элементы вне текста (например, список литературы, рисунки). Книга имеет собственный стиль, определяющий размер каждой страницы, расположение номеров страниц и колонтитулов на каждой странице, используемый шрифт, размещение таблиц и рисунков и т. д. Хотя раньше подобные тексты печатались на пишущих машинках, в настоящее время широкое распространение персональных компьютеров привело к росту популярности настольных издательских систем.

12.1.1. Подготовка документов в LaTeX

Пользователь настольной издательской системы полностью отвечает за все аспекты подготовки документа к печати. Для упрощения этого процесса были созданы

специальные языки обработки документов. Эта книга, например, была подготовлена в системе TEX (TeX), разработанной Дональдом Кнутом (Donald Knuth), с использованием макросов LATEX (LaTeX), разработанных Лесли Лэмпортом (Leslie Lamport). За неимением более подходящего термина мы часто говорим о компиляции книги, имея в виду обработку отдельных глав TEX-программой.

TEX-программа работает почти так же, как и традиционный компилятор. Как отмечалось в разделе 3.2, за первый проход TEX создает таблицу символов для отслеживания номеров разделов, страниц и рисунков. За второй проход генерируется выходной документ, в который вставляются правильные значения этих номеров.

Например, исходный текст с макросами LATEX для TEX-программы предыдущего абзаца выглядел следующим образом:

```
\TeX-программа работает почти так же. как и традиционный компилятор. Как отмечалось
в разделе \ref{translation.sec}, за первый проход \TeX создает таблицу символов
для отслеживания номеров разделов. страниц и рисунков. За второй проход генерируется
выходной документ. в который вставляются правильные значения этих номеров.
```

Макрос `\TeX` системы LATEX[1] печатает название TeX в несколько необычном виде: TEX; макрос `\ref` ссылается на предыдущий раздел книги, который в нашем случае помечен меткой с именем `translation.sec`, сохраненной в таблице символов при первом проходе компилятора TEX в виде значения 3.2. При втором проходе это значение метки используется для того, чтобы вставить в документ фактический номер раздела.

Но в отличие от традиционных концепций компиляции, связанных с хранением данных, вызовом подпрограмм и определением типов, программа реализации издательской системы заботится о компоновке страницы и разгонке набора. Например, для этой книги требуется трехпроходное выполнение TEX-программы. Во время первого прохода текст распределяется по строкам и страницам в соответствии с указанным стилем (либо встроенным в TEX, либо определенным в соответствии с руководством по созданию пользовательских стилей), а в таблицу символов заносятся различные ссылки, в том числе библиографические. Во время второго прохода эти ссылки размещаются в документе в тех местах, где они были упомянуты, и создается оглавление документа. Во время третьего прохода в документ в соответствующие места вставляются рисунки, оглавление и библиографические ссылки. Для документов без оглавления необходимо только два прохода, а для более сложных документов TEX может потребоваться большее, чем три, количество проходов, но два-три прохода являются типичным значением.

LATEX создает *окружения*, которые упрощают использование системы TTEX. Они действуют подобно правилам определения области видимости переменных в языках C или Pascal. Например, можно определять начало и конец списков, которые могут быть нумерованными или маркированными. Если начинается новый раздел или подраздел, то автоматически вставляются их соответствующие номера. Синтаксис LATEX аналогичен синтаксису блочно-структурированного языка программирования.

---

[1] В отечественной литературе часто систему TEX с макросами LATEX называют просто системой LATEX. — *Примеч. науч. ред.*

На рис. 12.1 представлена структура типичного документа, обрабатываемого системой LATEX. При вызове программы LATEX документ, содержащийся в файле latex.tex, считывается в TEX для создания команд, определяющих главы, разделы, подразделы, списки, таблицы, рисунки и другие многочисленные структуры, необходимые для написания простого документа. Команда \documentstyle (из LATEX) позволяет пользователю добавлять другие элементы стиля. Обязательный параметр article указывает на то, что следует прочитать и присоединить к файлу latex.tex файл article.sty с командами, необходимыми для оформления данного документа в виде статьи. Например, в *статьях* отсутствуют главы, но в стиле для создания книг book (то есть файле book.sty) главы определены. Опция 11pt определяет размер шрифта для текста документа (кегль 11) и предписывает прочитать встроенный стилевой файл art11.sty, содержащий дополнительную информацию о промежутках между строками и символами для текста, набираемого шрифтом кегля 11. Программа TEX вместе со стилевыми файлами article.sty и art11.sty формирует стандартный способ обработки LATEX-статей. Аналогичным образом в LATEX определены стили для отчетов, книг, слайдов и писем.

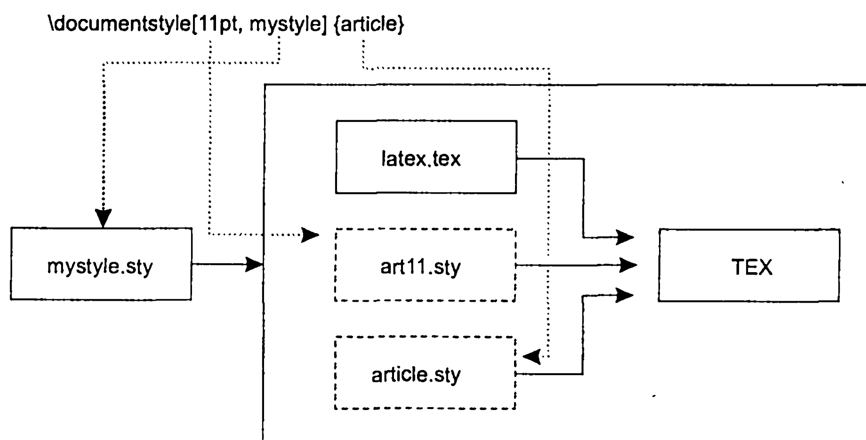


Рис. 12.1. Структура LATEX

У пользователя может возникнуть желание как-то изменить предлагаемые по умолчанию стили. Например, издательство Prentice Hall имеет собственный стиль для всех своих книг: верхний колонтитул на каждой странице включает номер текущей главы и раздела, размер страницы отличается от определенного по умолчанию в LATEX, промежутки между абзацами и разделами различны и т. д. Для уточнения стиля документа можно использовать необязательные опции команды \documentstyle, задаваемые в квадратных скобках. В нашем примере опция mystyle сообщает компилятору TEX, что необходимо прочесть файл mystyle.sty, в котором определены дополнительные стилевые команды, уточняющие правила форматирования книги.

Реализация. Системы, подобные TEX, компилируют документ для его выполнения на виртуальной машине текстового процессора. Обычно используется фор-

мат Postscript, который мы обсудим несколько позже. После компиляции документа выходной файл выполняется на виртуальной машине Postscript для просмотра или печати. Преимуществом такого подхода является то, что исходный документ — это текст, состоящий их символов ASCII, который можно просмотреть в любом текстовом редакторе.

LATEX использует для непосредственной компиляции документа стилевые файлы. В этом случае стиль документа и его текст могут существовать по отдельности. Подготовка доклада для конференции требует только добавления опции `twocolumn` к макросу `\documentstyle` для изменения форматирования документа в две колонки на странице вместо одной без каких-либо изменений в тексте самого документа. Такой подход позволяет использовать один текстовый файл для различных целей, если информация о стиле хранится отдельно. Но иногда при таком подходе бывает трудно понять, как же будет выглядеть окончательный документ.

12.1.2. WYSIWYG-редакторы

Альтернативным подходом к обработке документов являются WYSIWYG-редакторы (What You See Is What You Get — что видишь на экране, то и получишь при печати). В этом случае файл исходно форматируется так, как он будет выглядеть при печати. Все шрифты, интервалы, рисунки, таблицы, номера разделов сразу включаются в требуемые места. Все изменения в редактировании документа непосредственно отражаются на экране монитора. Это упрощает процесс редактирования, но усложняет реализацию изменения стилей. Такой подход к подготовке текстовых документов является наиболее распространенным в издательских системах на персональных компьютерах, например в Microsoft Word или WordPerfect фирмы Corel.

Реализация. В этом случае в документ встраиваются соответствующие команды редактирования и форматирования, а программой обработки текстов является виртуальная машина, разработанная для отображения результатов выполнения этих команд. Недостатком такого подхода является то, что каждая программа обработки текстов WYSIWYG определяет собственную архитектуру виртуальной машины. Поэтому файл, подготовленный для выполнения какой-либо одной программой обработки текстов, часто невозможно просмотреть с помощью другой программы.

Эту проблему частично решает *текстовый формат RTF*. RTF — это система обозначений для описания команд, выполняемых виртуальной машиной текстового процессора, использующая стандартную запись. Таким образом, любая система обработки текстов, в которой имеется встроенный транслятор из формата RTF в формат своей внутренней виртуальной машины, может обрабатывать документы, созданные другой программой WYSIWYG. Использование RTF значительно расширило возможности обмена документами между различными системами обработки текстов.

Языки описания страниц

Системы, подобные TEX, предназначены для написания документов. Существует другой класс текстовых языков — это языки описания страниц. К ним относятся

Postscript, созданный фирмой Adobe Systems, и HTML, который используется для описания страниц в технологии WWW в Интернете. В этом случае Postscript является, как правило, выходным языком систем, подобных TEX и используется для оформления текста на странице. В большинство принтеров Postscript встроены интерпретаторы Postscript, и документ Postscript действительно выполняется принтером, когда последний размещает текст и рисунки на странице. Как и TEX, Postscript имеет определенный синтаксис и может быть описан с помощью многих способов, обсуждавшихся в этой книге. HTML мы рассмотрим в деталях немного позже в этой же главе.

12.1.3. Postscript

Postscript был разработан Джоном Уорноком (John Warnock) и Чаком Гешке (Chuck Geshke) из Adobe Systems в начале 80-х гг. [7]. Исходно Postscript использовался как ядро механизма печати компьютеров Apple, но вскоре стал широко распространенным стандартом для большинства компьютерных систем. Интерпретаторы Postscript (в виде программных или аппаратных компонентов) для печати документов присутствуют практически во всех современных компьютерных системах.

В Postscript используется модель изображения текста (или рисунков) на чистой странице. Когда страница готова, она выводится на печать и начинается «прорисовка» изображения очередной страницы. Это есть не что иное, как метод компиляции, обсуждавшийся нами в начале этой главы в связи с классификацией трансляторов соответственно форме их вывода. Каждый документ Postscript включает в себя программу, которая печатает на принтере (или отображает на экране монитора) следующие друг за другом страницы.

Программа Postscript состоит из четырех компонентов:

1. *Интерпретатор для выполнения вычислений.* Основной моделью такого интерпретатора является простой стек постфиксного выполнения.
2. *Синтаксис языка.* Он основан на синтаксисе языка Forth, описанном в обзоре языка 8.2.
3. *Расширения для раскрашивания.* Расширение языка Forth командами закрашивания для управления процессом отображения текста и рисунков на листе бумаги.
4. *Соглашения.* Набор соглашений, не входящих в официальный язык Postscript, которые используют различные принтеры для согласования представления документов. Использование этих соглашений упрощает передачу документов Postscript из одной системы в другую.

Каждый компонент будет описан в последующих разделах.

12.1.4. Виртуальная машина Postscript

Программа Postscript состоит из последовательности команд, которые представляют постфиксную форму алгоритма, необходимого для прорисовки документа. Эта последовательность управляет стеком, как показано на рис. 12.2. Когда про-

грамма Postscript начинает выполняться, в стеке уже присутствуют два элемента, которые программа не может удалить:

- systemdict — это системный словарь, который представляет собой исходное связывание объектов Postscript с их внутренним представлением.
- userdict — это пользовательский словарь, который представляет собой новые определения, включенные в это выполнение программы Postscript. Сюда могут входить измененные определения элементарных объектов, уже определенных в systemdict.

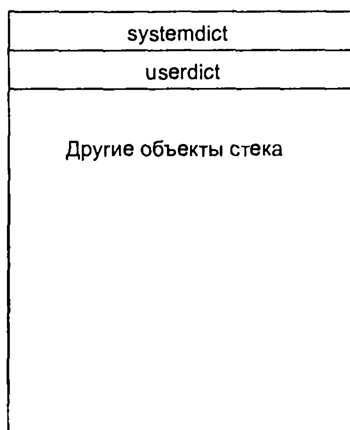


Рис. 12.2. Модель выполнения программы Postscript

Стеки выполнения. Фактически в Postscript используется четыре вида стеков:

1. Стек *операндов*, содержащий операнды, которые добавляются в стек, выполняются и затем удаляются из стека.
2. Стек *словарей* содержит только объекты-словари. Этот стек определяет область видимости и контекст каждого определения.
3. Стек *выполнения* содержит выполняемые объекты. По большей части это функции на промежуточных стадиях выполнения.
4. Стек *графики* управляет контекстом для отображения объектов на странице. Он рассматривается в следующем разделе.

Программа на Postscript записывается в виде последовательности символов ASCII. Программа разбивается на последовательность лексем. При считывании каждой лексемы ее определение становится доступным в стеке (сначала просматривается пользовательский словарь userdict, а затем системный systemdict), а затем лексема выполняется соответствующим действием.

Имена — это выполняемые объекты, обычно они состоят из любых символов, за исключением пробелов (например, AName, abc, A.Name). Вообще говоря, имена вызывают поиск их значений в стеке словарей с последующим выполнением найденных значений. Так, имя add приводит к доступности его определения в словаре systemdict. Это определение приводит к тому, что два верхних элемента в стеке операндов складываются. Конечно, пользователь может написать свое определение для имени add

и поместить его в словарь userdict, переопределив тем самым исходный примитив add. В листинге П.12 приведен пример простой программы на Postscript.

Литералы — это имена, перед которыми стоит символ наклонной черты (/). Этот символ служит указателем на то, что следующее за ним имя следует помещать в стек операндов, а не искать его значение в словарях. Использование наклонной черты позволяет *l*-значение add (то есть имя) поместить в стек операндов вместо того, чтобы вызвать *r*-значение (то есть процедуру сложения). Например, при определении новой функции до того, как это определение будет занесено в словарь, сначала нужно поместить имя и его определение в стеке операндов, а затем выполнить имя def. Имя def, когда будет найдено в системном словаре systemdict, приведет к тому, что два верхних элемента из стека операндов (которые должны представлять имя новой функции и ее выполняемое определение) будут добавлены в словарь userdict в качестве нового элемента.

Команды раскрашивания

До сих пор описание Postscript во многом напоминало описание языка Forth, который использует простую виртуальную машину постфиксных вычислений. Операнды располагаются в стеке, а операции используют содержимое стека в качестве своих аргументов. Силу языку Postscript придает возможность раскрашивать информацию на странице.

Имя show используется для размещения текста на странице, moveto перемещает курсор по странице, a showpage используется для отображения полностью готовой страницы. Предполагается, что изначально страница имеет белый цвет. Объекты, вычерчиваемые на странице, непрозрачны, то есть любой объект будет стирать объект, поверх которого он вычерчивается. Стирание части страницы осуществляется простой прорисовкой белого объекта в этой области.

Начало координат расположено в левом нижнем углу страницы, а точки на странице задаются с помощью координат (x, y): x соответствует расстоянию от начала координат по горизонтали, а y — по вертикали. Расстояние измеряется в пунктах; одному дюйму соответствует 72 пункта, но с помощью команды scale это можно изменить.

Пример рисунка. Чтобы продемонстрировать некоторые графические возможности Postscript, мы несколько расширим программу из листинга П.12 (которая является Postscript-версией программы на языке Forth из обзора языка 8.2) таким образом, чтобы она вычерчивала изображение простого грузовика. В листинге 12.1 приведена модифицированная программа, в которой строки с 14-й по 30-ю вставлены между строками 12 и 13 программы из листинга П.12.

box — это новая функция, которая создает прямоугольник со сторонами длиной 3 по горизонтали и 1 по вертикали. Заметим, что стороны этого прямоугольника равны соответственно 3 и 1 пункту, или 3/72 дюйма и 1/72 дюйма, которые слишком малы, чтобы можно было разглядеть прямоугольник на странице. Для увеличения прямоугольника используется команда масштабирования scale (строка 19). Команда newpath начинает новую траекторию. Курсор перемещается в начало координат командой moveto, а затем команда lineto используется для создания ряда линейных сегментов. Команда closepath используется для замыкания траектории, в результате чего получается замкнутая фигура.

Листинг 12.1. Программа вычерчивания 10-колесного грузовика

```
 0: %! Это файл Postscript
 1: %Аналогична програме на языке Forth
 2: /Helvetica findfont
 3: 20 scalefont
 4: setfont
 5: 200 400 moveto
 6: /formatit {10 10 string cvrs show} def
 7: /sqr {dup mul} def
 8: /dosum {exch 1 add exch 1 index sqr add} def
 9: 3 6 dosum 2 copy formatit ( ) show formatit
10: clear
11: 200 375 moveto
12: 0 0 0 1 9 {pop dosum} for formatit
14: %Рисуем грузовик
15: /box {newpath 0 0 moveto 0 1 lineto 3 1 lineto 3 0
16:    lineto closepath} def
17: .1 setlinewidth 0 setgray
18: gsave
19: 72 72 scale
20: 2 5 translate box stroke
21: 3.2 0 translate .5 .5 scale box fill
22: 0 1 translate .6 .6 scale box fill
23: grestore
24: /tire {newpath 1 0 moveto 0 0 1 0 360 arc closepath} def
25: .5 setlinewidth 10 10 scale
26: 16 34 translate tire stroke
27: 3 0 translate tire stroke
28: 17 0 translate tire stroke
29: 3 0 translate tire stroke
30: 8 0 translate tire stroke
13: showpage
```

Толщина линий устанавливается равной 0.1, а цвет задается при помощи команды setgray, которая с помощью своего параметра позволяет устанавливать различные оттенки серого цвета. Черному цвету соответствует параметр со значением 0, белому — 1[1]. В строке 18 сохраняются текущие графические настройки, чтобы их можно было восстановить позднее. Все дальнейшие изменения в масштабе снова устанавливаются в исходные значения, когда мы восстанавливаем текущее графическое состояние. Затем мы масштабируем координаты x и y так, чтобы один пункт соответствовал одному дюйму (72 пункта). Теперь прямоугольник со сторонами 1 × 3 пункта представляет прямоугольник со сторонами 1 × 3 дюйма.

Для размещения изображения грузовика на странице (строка 20) мы переносим начало координат на 2 пункта (что теперь соответствует 2 дюймам) по горизонтали и на 5 пунктов по вертикали. Теперь начало координат (0,0) находится рядом с числами, напечатанными в результате выполнения первых двенадцати строк программы. Последовательность команд box stroke вычерчивает прямоугольник размером 3 × 1, левая нижняя вершина которого расположена в новом начале координат, и закрашивает линейные сегменты, представляющие стороны прямоугольника. (Мы также могли бы использовать команду fill для закрашивания внутренней области вычерченного прямоугольника, что мы и демонстрируем в следу-

[1] Промежуточные значения соответствуют различным оттенкам серого цвета. — *Примеч. науч. ред.*

ющей строке.) Затем мы последовательно перемещаем начало координат для вычерчивания кабины грузовика и его колес. В строке 24 определяется команда `tire`, которая рисует колесо грузовика в виде окружности.

Полностью программа приведена в листинге 12.1, а вычерчиваемая ею страница показана на рис. 12.3. Обратите внимание на то, что в начале программы добавлена строка комментария, начинающаяся с последовательности символов `%!`. Этот комментарий является соглашением Postscript (см. конец раздела 12.1), и он часто бывает необходим для сообщения программам (например, печатающим документы), что файл на самом деле представляет собой программу Postscript.

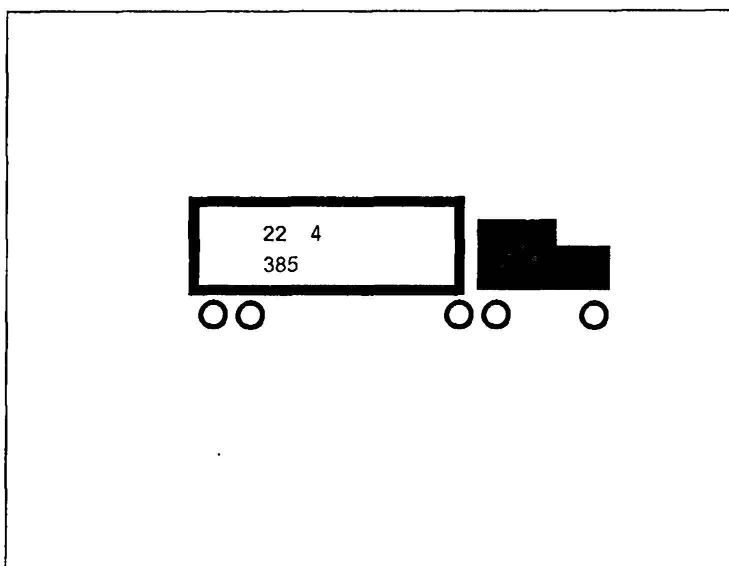


Рис. 12.3. Великолепное художественное произведение авторов

Команды закрашивания областей. Одной из наиболее интересных особенностей Postscript является возможность закрашивать замкнутые фигуры в соответствии с шаблоном. Мы уже использовали команду `box fill` (строка 21 в программе из листинга 12.1), но возникает довольно сложный вопрос: как определить, когда мы находимся внутри данной фигуры, а когда — снаружи? Реализованы два алгоритма.

1. *Правило ненулевой намотки.* Этот метод используется операцией `fill`. Проводится линия из точки в бесконечность. Для каждого сегмента границы фигуры, который пересекает линию слева направо, добавляем 1; для каждого сегмента, который пересекает эту линию справа налево, вычитаем 1. Если в сумме получится ноль, то данная точка лежит вне фигуры; в противном случае точка расположена внутри фигуры.

2. *Правило четности и нечетности.* Этот алгоритм используется операцией `eofill`. Проводится линия из точки в бесконечность. Если она пересекает сегменты границы фигуры нечетное количество раз, то точка находится внутри; если же количество пересечений четное, то эта точка расположена вне фигуры.

Различие между двумя операциями, закрашивающими фигуры, показано на рис. 12.4. Звезда определяется с помощью процедуры

```
/star {newpath 1 0 moveto 6 4 lineto 0 4 lineto 5 0 lineto
      3 6 lineto closepath} def
```

Левая звезда вычерчена с помощью команд `star stroke`, центральная — с помощью `star fill` и правая — с помощью `star eofill`.

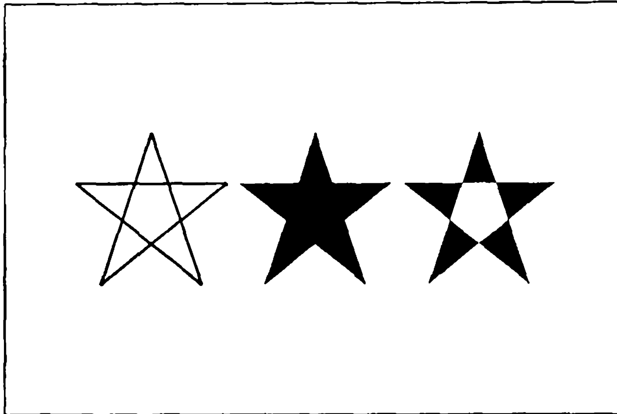


Рис. 12.4. Команды *stroke*, *fill* и *eofill*

Соглашения в документах Postscript

То, с чем мы уже познакомились в Postscript (см. конец раздела 12.1.4), достаточно для написания программ, создающих интересные документы. Однако в Postscript существуют некоторые соглашения, которые часто используются и иногда требуются в различных виртуальных машинах Postscript. Обычно эти соглашения имеют вид комментариев, которые могут игнорироваться, но часто передают информацию интерпретатору Postscript. Обычно они располагаются в начале файла.

Наиболее важным соглашением является тот факт, что первый комментарий должен начинаться с последовательности `%!`. Этот комментарий сообщает интерпретатору, что данный файл является Postscript-программой. Большинство программ, генерирующих файлы Postscript, помещают в этой строке свои названия. Минимально согласующаяся программа содержит `%!PS` в своей первой строке. Каждая страница документа обычно начинается командой `save` и заканчивается командой `restore`, чтобы изолировать ее от эффектов, определенных для других страниц. Определяющие структуру документа соглашения задаются и продолжаются в строках, начинающихся с символов `%%`, пока не встретится первая строка без этих символов. Некоторые из этих соглашений приводятся ниже.

- `DocumentFonts`: список шрифтов, используемых в документе.
- `Title`: произвольная строка, определяющая заголовок документа.
- `Creator`: имя человека или программы, создавшей данный файл.
- `CreationDate`: дата и время создания файла.
- `Pages`: количество страниц в документе.

- BoundingBox: четыре числа, которые определяют координаты левого нижнего и правого верхнего углов страницы, которая реально заполняется программой. Это соглашение позволяет вставлять страницы в другие документы.

Краткий обзор Postscript. Postscript был разработан как архитектура виртуальной машины, предназначенной для создания печатных документов. В большинстве приложений не предполагается, что программист будет читать текст документа Postscript. Тем не менее синтаксис Postscript достаточно прост и легок для восприятия. Существуют образовательные программы для обучения этому языку программирования. Его синтаксис и семантика отличаются простотой, а доступность программ для отображения на экране документов Postscript означает, что у любого пользователя имеется возможность доступа к интерпретатору виртуальной машины, на которой можно тестировать свои Postscript-программы.

Следующим этапом развития Postscript стало создание фирмой Adobe формата PDF (Portable Document Format — формат переносимых документов). PDF — это форма сжатия файлов Postscript. Программы чтения PDF-файлов свободно распространяются по Интернету, а большинство web-браузеров могут отображать PDF-файлы. Формат PDF повсеместно используется для передачи и отображения форматированных документов. Бесплатное распространение программ чтения PDF-файлов можно считать тонким ходом фирмы Adobe, так как теперь она продает программу Acrobat, необходимую для создания файлов в формате PDF.

12.2. Всемирная паутина WWW

Широкое распространение Интернета и рост популярности такого занятия, как *«путешествие» по сети* привели к повышению интереса к языкам семантического описания, упомянутых в начале этой главы. В этом разделе мы расскажем о том, как был разработан Интернет и откуда он пришел. Мы предоставим краткий обзор роли языка Java в развитии Интернета. Напоследок мы обсудим язык разметки HTML и ту роль, которую он играет в отображении информации и навигации по Всемирной паутине WWW.

12.2.1. Интернет

В течение 60-х гг. концепция пакетной обработки данных уступила место интерактивной модели взаимодействия с компьютером с помощью терминалов (см. раздел 1.2.2). По мере того как мощность машин возрастала, появлялось желание организовать совместное использование информации пользователями различных систем.

В конце 60-х гг. Управление перспективных исследовательских программ Министерства обороны США (Advanced Research Project Agency, ARPA) начало разработку проекта, целью которого было выяснить, возможно ли организовать такую связь между компьютерами, разделенными достаточно большим расстоянием, которая позволила бы пользователю, работающему за терминалом одного компьютера, получать доступ к ресурсам (то есть программам и данным), находящимся на другом компьютере. Этот проект положил начало раэвитию сети

ARPANET. Подробный рассказ об ARPANET не входит в задачи авторов данной книги. Достаточно будет сказать, что главную трудность представляла собой невозможность наладить надежную пересылку сообщений от одного компьютера к другому.

Начало сети ARPANET датируется 1970 г., когда была создана сеть, соединявшая три узла BBN в Кембридже (штат Массачусетс), Калифорнийский университет в Лос-Анджелесе (UCLA) и SRI в Калифорнии и использовавшая линии связи с пропускной способность 56 кбит/с. С течением времени к сети добавлялись другие университеты и военные организации, так что к середине 70-х гг. сеть насчитывала уже несколько сотен узлов.

Связь между двумя компьютерами осуществлялась с помощью *сообщений*. Сообщение разбивалось на строки фиксированной длины, называемые *пакетами*, и эти пакеты пересылались один за другим, пока все сообщение не оказывалось переданным на принимающий компьютер. Чтобы гарантировать надежную доставку сообщений до компьютера назначения, была разработана формальная модель передачи сообщений, называемая *протокол*. Для ARPANET был разработан протокол управления передачей – протокол Интернета (Transmission Control Protocol/Internet Protocol, TCP/IP).

TCP/IP являлся механизмом связи низкого уровня, который просто определял, что последовательность байтов, предназначенная для конкретного компьютера, дошла до него без изменений, в своем исходном виде. Как правило, для пользователей этот механизм был слишком сложным, чтобы они могли его использовать непосредственно для доступа к удаленному компьютеру. Поэтому были разработаны другие протоколы, которые пользователи могли вызывать; эти новые протоколы были реализованы в программах, которые фактически передавали данные, используя TCP/IP. В сети ARPANET широко использовались три протокола.

- *Telnet* – это протокол, который позволяет пользователю одного компьютера подключаться к другому, удаленному компьютеру и работать с ним, как через обычный терминал. Используя терминологию распределенного программирования, можно сказать, что пользователь работает за клиентским компьютером, который функционирует как терминал, а терминальная программа связывается с помощью протокола telnet с удаленным главным (host) компьютером, на котором работает серверная программа. Это позволяет пользователю войти в систему компьютера, расположенного в совершенно другом месте, при этом удаленный компьютер будет воспринимать этого пользователя как локального. С помощью протокола telnet пользователь, работая за компьютером в одном университете, мог зарегистрироваться на компьютере другого университета и использовать все возможности этого удаленного компьютера так же, как если бы он работал на нем непосредственно.
- *SMTP* – это простой протокол электронной почты (Simple Mail Transfer Protocol). Он является основой электронной почты – широко распространенного в наше время способа общения. В каждой почтовой программе (например, Berkley mail, Microsoft Outlook, Eudora, Lotus Notes) имеются свои собственные механизмы для создания почтовых сообщений, сохранения их в файловой системе пользователя, а также для обработки дополнительных

команд. Тем не менее все почтовые программы компилируют сообщения в один и тот же формат, который определяется протоколом SMTP. Затем почтовая программа отправляет письмо по Интернету, и описанный ранее набор протоколов TCP/IP гарантирует, что письмо дойдет до места назначения. Строгое соблюдение протокола SMTP — это то, что позволяет электронной почте функционировать столь успешно. Электронное сообщение может отправить кто угодно, и получатель сможет прочитать его, используя практически любую почтовую программу. Здесь наблюдается сильный контраст с тем хаосом, который царит в области программ обработки текстов, — документ, созданный в какой-либо одной программе, бывает весьма непросто прочитать в другой.

- *FTP* — протокол передачи файлов (File Transfer Protocol). В течение многих лет этот протокол являлся стандартным способом получения данных с удаленного компьютера. FTP предоставлял механизм перемещения файла с одного компьютера на другой. При этом требовалось вызвать программу FTP-клиент на локальной машине, при помощи протокола FTP подсоединиться к удаленной сервер-машине и затем получить нужные документы с этой машины или переслать документы с локальной машины пользователя на удаленную. Для того чтобы документы могли быть доступны любому пользователю, FTP позволял *анонимный* вход в систему удаленного компьютера. Пользователи могли с помощью FTP связаться с удаленным компьютером, используя специальное пользовательское имя *anonymous*, для которого не требовался ввод пароля, после чего они имели доступ к документам из определенного каталога файлов на удаленном компьютере. Этот механизм был одним из первых способов распространения информации любому заинтересованному пользователю.

Слабости в механизме FTP очевидны. Прежде всего пользователь должен был знать, на какой машине находится интересующая его информация. Далее, у него должно быть право доступа к файлам на этой машине с интересующей его информацией. Частично эта проблема решалась с помощью анонимного входа в систему. Далее, пользователь должен был точно знать, где в файловой системе находятся необходимые ему документы. Несмотря на все эти недостатки, долгие годы протокол FTP являлся стандартным механизмом передачи информации, пока HTML не изменил ситуацию в корне. HTML мы обсудим подробнее несколько позже.

В середине 80-х гг. управление APRA приняло решение прекратить поддержку APRANET. В это время уже было ясно, что концепция оправдала себя, но APRA, в силу своей исследовательской специфики, не могло заниматься организацией коммерческих услуг по доступу к сети. К этому времени сеть ARPANET объединяла несколько тысяч компьютеров по всему миру. Академические и военные структуры, а также компании, занятые в высокотехнологичном производстве, интенсивно использовали возможности, предоставляемые этой сетью. В США Национальный научный фонд (National Science Foundation, NSF) взял на себя поддержку костяка сети — высокоскоростных телефонных линий, которые обеспечивали основной трафик соединений с помощью TCP/IP хост-компьютеров. Название сети постепенно трансформировалось в Интернет. К костяку сети добавлялись локальные сети штата, университета, компании, пока Интернет не стал аморфным объедине-

нием компьютеров, постоянно взаимодействующих друг с другом. Коммерческие провайдеры, называемые теперь Интернет-провайдерами (Internet Service Providers, ISP) (например, Microsoft Network, Compuserve, AOL), установили связи с Интернетом, чтобы индивидуальные пользователи могли входить в сеть со своих домашних компьютеров, используя модем для подключения к своим локальным провайдерам. В настоящее время никто не может точно определить, сколько миллионов пользователей имеют доступ к Интернету. Подключившись к Интернету, пользователи могли использовать имеющиеся в то время в их распоряжении протоколы (SMTP, FTP, Telnet) для пересылки сообщений электронной почты, получения файлов и входа на удаленные хост-компьютеры.

В начале 90-х гг. фонд NTF также решил отказаться от своей роли в сетевом бизнесе, и управление сетью перешло к различным коммерческим компаниям. Развитие Интернета продолжалось, и теперь сеть объединяет миллионы пользователей. Точное количество определить невозможно, так как централизованный контроль ресурсов Интернета отсутствует. Скорость передачи информации возросла с 56 кбит/с до 1 Мбит/с и даже 100 млн бит/с (по оптоволоконным линиям), хотя для индивидуальных пользователей, которые подключаются к сети со своих домашних компьютеров, скорость передачи обычно не превышает 33,6 или 56 кбит/с, что обусловлено возможностями используемых модемов. Одним из современных способов увеличения скорости передачи информации для пользователей домашних компьютеров является использование линий кабельного телевидения, которые позволяют увеличить скорость до величин порядка мегабит в секунду.

Создание WWW

К концу 80-х гг. значительно возрос интерес к поиску более легких способов передачи файлов от одного удаленного компьютера другому. Как уже было сказано, протокол FTP являлся не слишком удачным решением этой задачи. Технология WWW (или Web) была разработана в виде дополнительных протоколов к трем уже широко используемым протоколам (SMTP, FTP, Telnet), упоминавшимся ранее. Физики — главным образом Тим Бернерс-Ли (Tim Berners-Lee) из исследовательской лаборатории физики частиц высокой энергии Европейской лаборатории ядерных исследований (Conseil Europeen pour la Recherche Nucleaire, CERN), расположенной в Швейцарии недалеко от Женевы, — задались целью создать более простой, чем стандартный FTP-сервер, механизм доступа и передачи документов с удаленного компьютера. Они разработали концепцию семантического описания, которая приведена ранее в этой главе. Одна, серверная, программа отображала документ, а другая, клиентская программа, называемая *браузером*, читала и расшифровывала этот отображенный документ. Сила разработанной ими системы заключалась в том, что отображаемый документ содержал указатели на другие документы — то, что в кругах специалистов по вычислительной технике получило название *гипертекст*. Более ранней версией гипертекста был продукт HyperCard компании Apple для компьютеров Macintosh, но разработка CERN была большим достижением в том отношении, что гипертекст позволял связываться с документами, расположенными на других компьютерах, подключенных к Интернету.

Разработанный в CERN протокол был протоколом передачи гипертекста (Hypertext Transfer Protocol, HTTP), а каждый указатель на другой документ стал известен как универсальный указатель ресурса (Uniform Resource Locator, URL). Процедура доступа к месту расположения документа сократилась до следующих шагов: запуск web-браузера на локальной машине, ввод адреса URL искомого документа, соединение с web-сервером, функционирующим на удаленной машине, содержащей документ с заданным URL, и отображение документа в соответствии с протоколом HTTP. Щелкая левой кнопкой мыши на внедренных в отображаемый документ в виде гиперссылок адресах URL, пользователь мог переходить с одного web-сервера на другой.

Этот способ получения информации имел большие преимущества перед механизмом FTP, который использовался ранее. Web-браузеры полностью изменили природу Интернета. В 1993 г. Национальный центр по использованию суперкомпьютеров (National Center for Supercomputing Applications, NCSA) выпустил web-браузер Mosaic. Этот браузер предоставлял интерфейс, с помощью которого любой пользователь с легкостью мог переходить от одного документа к другому простым щелчком кнопкой мыши на гиперссылке, внедренной в документ. Теперь Интернет перестал быть инструментом академических и исследовательских кругов, и любой пользователь мог использовать его для доступа и получения информации. Если браузеры и отображаемые документы соответствуют протоколу HTTP, любой браузер может взаимодействовать с любым web-сервером. Это позволило Интернету превратиться из инструмента, доступного лишь опытным пользователям, в широко распространенное приложение, открытое для любого пользователя.

На рис. 12.5 приведена схема, иллюстрирующая архитектуру WWW с промежуточным звеном. Предположим, вам нужно найти в сети информацию об этой книге[1], которая расположена на сайте издательства «Прентис-Холл» (Prentice-Hall). Для этого потребуется выполнить следующие действия.

1. Ввести URL-адрес *домашней страницы* — страницы, содержащей информацию о каком-либо человеке или об организации. В нашем случае http://www.cs.umd.edu/users/mvz/pzbook. Этот адрес состоит из двух частей: *доменного имени* www.cs.umd.edu, то есть имени машины, содержащей искомую web-страницу, и файла на этой машине, который и представляет требуемую web-страницу с информацией об этой книге users/mvz/pzbook.
2. web-браузер посылает доменное имя одной из нескольких специальных Интернет-машин, называемых серверами доменных имен (Domain Name Server, DNS). DNS по заданному доменному имени компьютера возвращает его IP-адрес (Internet Protocol — протокол Интернета). Каждая машина в сети имеет свое уникальное доменное имя и свой уникальный IP-адрес. IP-адрес — это последовательность четырех байтов из восьми битов, которая обычно записывается в виде четырех десятичных чисел, разделенных точками. (В нашем примере введенному доменному имени соответствует IP-адрес 128.8.128.80.)

[1] Имеется в виду оригинальное издание на английском языке. — *Примеч. пер.*

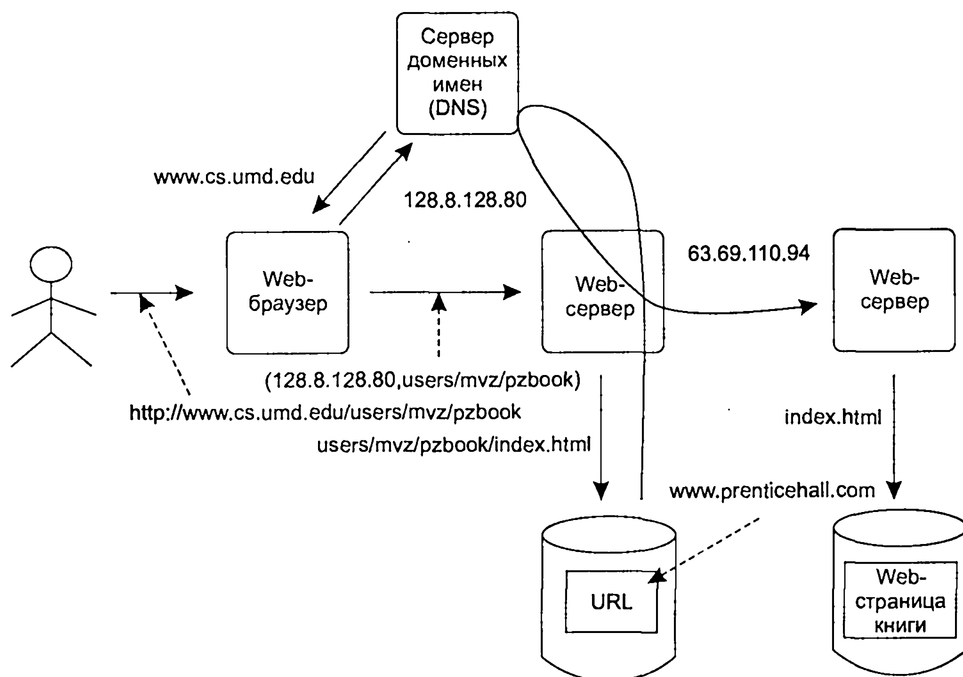


Рис. 12.5. Организация доступа к web-страницам

3. Web-браузер посылает имя файла (users/mvz/pzbook) web-серверу с IP-адресом 128.8.128.80. Программа HTTP-демон (HTTPD) на этой машине является основным средством взаимодействия между сервером и Интернетом. Эта программа постоянно осуществляет мониторинг подсоединения компьютера к сети Интернет, отслеживая соответствующие HTTP-сообщения. Демон — это программа операционной системы, которая постоянно выполняется. Программы печати, программы управления памятью и программы-планировщики являются типичными программами-демонами в большинстве операционных систем. Чтобы реализовать web-технологию, на компьютере-сервере должен быть установлен демон HTTP.
4. В нашем примере web-сервер к имени users/mvz/pzbook добавит имя index.html, так как заданный нами файл users/mvz/pzbook являлся каталогом, а не файлом. (Сервер на персональном компьютере обычно добавляет index.htm или default.htm.)
5. Содержимое файла users/mvz/pzbook/index.html отсылается назад web-браузеру и отображается для пользователя.
6. Если теперь пользователь щелкнет на гиперссылке с URL-адресом издательства «Прентис-Холл» (www.prenticehall.com), которая содержится на отображенной странице, то процесс повторится заново, браузер соединится с web-сервером издательства «Прентис-Холл» по IP-адресу 63.69.110.94, после чего на компьютере пользователя отобразится соответствующая web-страница.

Аналогичным образом пользователь может переходить по ссылкам с одной страницы на другую, посещая web-серверы по всему миру, пока не найдет требуемую информацию.

Порталы. Чтобы облегчить навигацию в Интернете, теперь в сети существуют web-сайты, называемые *порталами*, — сайты входа во Всемирную паутину WWW. На этих сайтах имеются программы, известные как *поисковые машины*. Поисковая машина — это обработчик запросов, который, получив запрос пользователя, возвращает список соответствующих web-страниц в виде гиперссылок. Щелкая мышью на подходящих ссылках, пользователь может просматривать web-страницы, получая информацию.

Поисковые машины часто действуют как *web-навигаторы*. Начиная с какой-либо страницы, web-навигатор перебирает все имеющиеся на странице ссылки. Таким образом, в принципе, можно добраться до любой странице в сети, даже не зная ее URL. Web-навигатор строит базу данных кратких рефератов всех страниц, и запрос к поисковой машине возвращает информацию из этой базы данных.

Например, в каком-либо университете у каждого профессора может иметься домашняя страница. На такой странице могут содержаться ссылки на домашнюю страницу факультета и университета. Те университеты, в которых профессоры получили свои научные степени, также могут быть перечислены на их домашних страницах, как и компании и организации, с которыми они сотрудничают. Используя этот постепенно расширяющийся круг ссылок, можно, в принципе, достигнуть любой точки в Интернете. Различие между разными поисковыми машинами заключается в том, каким образом в них классифицируется информация, полученная в ответ на запрос. Алгоритм классификации в основном является закрытой информацией, поэтому при использовании разных поисковых машин для ответа на один и тот же вопрос можно получить очень разные ответы.

HTML

Одной из важных сторон навигации во Всемирной паутине является возможность отображения web-страниц с web-сервера и нахождения и активизации URL-адресов при переходе от одного web-сайта к другому. HTML — это язык семантического описания, разработанный уже упоминавшейся группой из CERN под руководством Тима Бернерса-Ли. Он создает виртуальную машину, на выполнение которой запрограммирован web-браузер при осуществлении web-функций. Что касается фактических деталей системы обозначений для создания этого нового протокола, в CERN было принято решение использовать версию стандартного обобщенного языка разметки (Standard Generalized Markup Language, SGML), которую назвали HTML (HyperText Markup Language — язык разметки гипертекста).

SGML. Сначала мы обсудим общую систему обозначений SGML, а затем перейдем к рассмотрению HTML как экземпляра SGML.

История SGML. Развитие SGML началось в конце 60-х гг., когда был создан язык для форматирования документов с помощью тегов-описаний. Вместо использования собственных ключевых слов документ мог быть структурирован в соответствии с некоторым заранее определенным множеством терминов, например заголовок раздела, название, рисунок и т. д. В таком контексте семантическая информация о документе могла бы передаваться вместе с его фактическим содержанием.

В конце 60-х гг. Комитет оформления ассоциации графической информации (Graphic Communication Association Composition Committee), используя идеи Вильяма Танниклифа (William Tunnicliffe), Стенли Райса (Stanley Rice) и Нормана Шарпфа (Norman Scharpf), разработала концепцию определения различных общих кодов, необходимых для различных документов, получившую название GenCode(R). В 1969 г. Чарльз Гольдфарб (Charles Goldfarb) из IBM совместно с Эдвардом Мошером (Edward Mosher) и Раймондом Лори (Raymond Lorie) разработали обобщенный язык разметки (Generalized Markup language, GML) идентификации компонентов документа с помощью специального вложенного синтаксиса. В 1978 г. Национальный институт стандартизации США (American National Standards Institute, ANSI) организовал специальную группу стандартов для объединения концепций GenCode(R) и GML. Результатом этой работы стал SGML, который был утвержден Международной организацией по стандартам (International Standards Organization, ISO) в 1986 г. под номером 8879.

Синтаксис SGML. Документ состоит из текста, который является неструктурированной последовательностью символов. Но в текст могут быть встроены *элементы SGML*. Семантика этих элементов не специфицирована, но их синтаксис определен. Элементы заключены между начальным тегом и завершающим тегом. Например, элемент *zork* начинается с тега <zork> и завершается тегом </zork>. Весь текст, расположенный между тегами, принадлежит этому элементу. Например:

```
<zork> I am a zork </zork>
```

определяет строку I am a zork как содержание элемента zork.

В тех случаях, когда конец элемента определяется однозначно, завершающий тег можно опустить. Например, если бы мы определили доклад как документ SGML, состоящий из названия (title), имени автора (author), краткого содержания (abstract) и тела доклада (body of text), мы могли бы дать его описание в виде

```
<report>
<title> текст </title>
<author> текст </author>
<abstract> текст </abstract>
<body> текст </body>
</report>
```

или

```
<report>
<title> текст
<author> текст
<abstract> текст
<body> текст
</report>
```

поскольку каждый раздел однозначно переходит в следующий.

Если мы определяем тело доклада как последовательность разделов, каждый из которых содержит вложенные разделы, до неограниченной глубины, тогда каждый раздел определяется с помощью синтаксиса:

```
<section> текст </section>
```

В данном случае, однако, необходимо использовать завершающий тег, так как местоположение границы между вложенными разделами однозначно не определяется.

Важной особенностью SGML является то, что он отделяет атрибуты документа от его содержания. Мы могли бы напечатать элемент <report> любым шрифтом и не разрушить основной смысл документа. Мы могли бы напечатать элемент <title> шрифтом 20-го кегля, а элемент <author> — шрифтом 16-го кегля или вообще все элементы напечатать шрифтом 10-го кегля. Все это никак не повлияет на контекст документа. Как мы увидим позже, в HTML это связано с некоторыми проблемами, так как файл HTML является одновременно и описанием того, что содержит web-страница, и описанием того, как эта страница будет отображена.

Применение SGML может решить более серьезные проблемы, касающиеся представления документа. Например, даты в разных странах часто отображаются в различных форматах (см. раздел 1.3.4). Сообщение (memo) «Встреча состоится 11/10/01» в Европе и в Соединенных Штатах будет воспринято по-разному. Если это сообщение распространяется среди сотрудников международной компании, то может возникнуть недоразумение — часть сотрудников может подумать, что встреча запланирована на 10 ноября, а часть решит, что имеется в виду 11 октября. Но если бы это сообщение было составлено с использованием SGML, то можно было бы использовать соответствующий элемент <date> и записать его в виде .

```
<memo> Встреча состоится
<!Это комментарий SGML — оформление даты>
<date> <month> 11 <day> 10 <year> 2001 </date>.
</memo>
```

Различные клиентские почтовые программы в Европе и в США отобразили бы элемент <date> нужным образом для каждого пользователя.

Множество используемых элементов описывается в компоненте SGML, называемом объявлением типа документа (Document Type Declaration, DTD), в котором содержится грамматическое описание всех доступных для данного документа тегов. Хотя в SGML имеются и другие компоненты, приведенного здесь описания достаточно, чтобы перейти к рассмотрению HTML.

Синтаксис и семантика HTML. Когда группа из CERN разрабатывала исходную концепцию браузера просмотра гипертекстовых документов, нотация SGML была взята за основу нотации и нового языка, который получил название язык разметки гипертекста (HyperText Markup Language, HTML). Имена файлов, содержащих текст HTML, выглядят как name.html. (На PC, где имеется ограничение на длину расширения имени файла, которое не должно превышать трех символов, HTML-файлам обычно даются имена name.htm.) По умолчанию, когда web-браузер пытается получить доступ к новому компьютеру, он читает HTML-файл с именем index.html. Например, если вы хотите получить обновленную информацию об этой книге, вам нужно ввести URL-адрес ее домашней страницы, созданной авторами, на сайте университета Мэриленда:

http://www.cs.umd.edu/users/mvz/pzbook/

и тогда ваш web-браузер по умолчанию будет искать в указанном каталоге файл index.html, а затем отобразит его.

Файлы HTML соответствуют приведенному выше синтаксису SGML. Минимальный документ HTML имеет следующий синтаксис:

```
<html>
<title> Заглавие документа </title>
<body> Текст документа </body>
</html>
```

Регистр в тегах не имеет значения. Таким образом, <html> и <HTML> означают одно и то же. Помимо этого в теги иногда могут входить некоторые необязательные параметры (атрибуты).

Текст тела HTML-документа может иметь дополнительную структуру. Некоторые наиболее важные элементы HTML перечислены ниже.

1. *Теги разделов.* Они разделяют части документа. <h1> — это заголовок первого уровня. Обычно он печатается крупными буквами. Обратите внимание на то, что этот тег <h1> объединяет семантическое содержание элемента (заголовок первого уровня документа) и его представление (крупный шрифт). Поскольку web-браузеры имеют достаточную свободу при отображении тегов HTML, одна и та же web-страница может выглядеть по-разному в различных браузерах.

   Теги <h2> и <h3> используются для задания подзаголовков документа (заголовков второго и третьего уровней соответственно) и отображаются шрифтами меньшего размера. Тег <hr> вычерчивает горизонтальную линию в документе, а его атрибут width (например, <hr width = 50%>) определяет ее длину. Тег <p> начинает новый абзац, а тег
 (от английского *break* — разрыв) начинает новую строку без создания нового абзаца.

2. *Теги представления.* Они формируют внешний вид документа при его отображении в браузере. Однако как именно они отображаются, в языке в точности не определено, поэтому браузеры имеют широкую свободу в выборе формата их представления. Тег <b> используется для отображения текста полужирным шрифтом, а тег <i> — для выделения текста курсивом. Тег <blink> отображает мерцающий текст (это обычно очень раздражает пользователя, поэтому такой способ выделения текста лучше не применять). Тег <pre> используется для того, чтобы отобразить текст точно в том формате, какой был использован в файле HTML, со всеми интервалами и символами новой строки. Этот тег был задуман как попытка обеспечить стандартизированный формат, независимый от браузера. Вообще, HTML игнорирует пробельные символы при форматировании текста для его отображения.

   Тег <center> центрирует текст, а тег <blockquote> используется для форматирования длинной цитаты.

3. *Теги списков.* Списки элементов могут отображаться различными способами: тег <li> начинает следующий элемент списка, который завершается либо концом самого списка, либо началом следующего элемента списка (то есть закрывающий тег отсутствует). Списки могут быть нумерованными, задаваемые тегом <ol> (ordered list — упорядоченный список), или неупорядоченными, задаваемыми тегом <ul> (unordered list — неупорядоченный список). Списки определений (тег <dl>) имеют встроенные теги <dt> для определяемых терминов, а сами определения задаются тегами <dd>.

4. *Гиперссылки.* Сила HTML обусловлена его ссылками на другие ресурсы HTML. Концепция URL (унифицированного указателя ресурса) является

тем механизмом, который осуществляет эту связь. С помощью специальных тегов документа HTML можно организовать связь с другим документом HTML с помощью его URL. В общем случае синтаксис URL имеет следующий вид:

http://имя_компьютера/путь_к_файлу/имя_файла

где:

- http: означает, что доступ к ресурсу осуществляется в соответствии с протоколом HTTP;
- // указывает, что этот адрес нужно искать не на локальной машине;
- имя_компьютера — это доменное имя компьютера в Интернете. Сервер доменных имен — это машина, которая переводит доменное имя в IP-адрес, указывающий точное расположение машины в Интернете;
- путь_к_файлу/имя_файла — это полное имя файла с указанием пути доступа к нему на машине, определяемой доменным именем.

Ссылка на другой документ HTML задается тегом <a>. Например, в элементе <a HREF = url-адрес> текст *текст* будет каким-либо образом выделен web-браузером (это зависит от конкретного браузера). Щелчок кнопкой мыши на выделенном тексте приводит к тому, что браузер соединится с файлом HTML, расположенным в сети по указанному URL-адресу, и отобразит его. Описанный механизм лежит в основе путешествий по сети.

5. *Графические изображения.* Тег <IMG SRC = адрес_файла ALT = текст> отображает на экране рисунок, находящийся по указанному адресу. Значением атрибута ALT является *текст*, который должен быть отображен, если браузер по какой-либо причине не может отобразить этот рисунок. Атрибут align, который может принимать значения top, middle и bottom, предназначен для выравнивания следующего за рисунком текста по верхней границе, середине или нижней границе рисунка соответственно. Очень существенное ограничение заключается в том, что следом за рисунком может расположиться только одна строка текста. В HTML-документах обычно используются рисунки формата GIF или JPEG.

При определении URL-адреса с помощью атрибута HREF тега <a> мы указывали на использование протокола HTTP, задавая его как часть адреса в виде http:. Однако этот протокол — всего лишь один из возможных протоколов, которые могут использоваться для доставки информации в Интернете. В общем случае URL-адрес имеет следующий вид:

протокол://имя_компьютера/путь_к_файлу/имя_файла

где определяется протокол, с помощью которого будет доставлен указанный ресурс. HTML-файлы, доставляемые с помощью протокола HTTP, — это всего лишь один из информационных ресурсов, на которые могут указывать гиперссылки в документе. Можно организовать связи и с другими ресурсами через другие протоколы.

- *FTP* — доступ к ресурсу с указанным URL осуществляется с помощью протокола FTP. Этот протокол используется для передачи файлов браузеру, особенно файлов, формат которых отличен от HTML.

- *GOPHER* – доступ к ресурсу с указанным URL осуществляется с помощью протокола Gopher, который является одним из ранних протоколов, использовавшихся для доступа к информации в Интернете. Протокол Gopher только начал завоевывать популярность, когда появился HTML и занял его место.
- *MAILTO* – в данном случае URL является адресом электронной почты. Если пользователь щелкает мышью на этом URL, то web-сервер загружает почтовую программу для создания почтового сообщения, которое отправляется по указанному адресу.
- *FILE* – определенный здесь URL является адресом файла на локальной машине. В зависимости от расширения имени файла вызывается соответствующая программа для его просмотра. Например, если имя файла – report.ps, то для его просмотра будет вызвана программа Postscript, picture.tif откроет программу просмотра графических файлов в формате TIFF, sound.au – это аудиофайл и т. д. Это позволяет web-браузеру иметь полный доступ ко всей файловой системе на локальном компьютере пользователя.
- *NEWS* – URL news:группа запускает локальный сервер новостей для доступа к указанной группе новостей.

Одним из недостатков WWW является то, что каждый браузер должен знать о каждом из перечисленных протоколов. Если добавляется какой-либо новый протокол, то все браузеры требуется обновить, чтобы они имели возможность работать с новым протоколом. Учитывая бессистемность и несколько хаотический рост Интернета, этот недостаток может, в принципе, привести к весьма нежелательным последствиям. Далее мы обсудим язык Java, который был разработан специально для решения этой проблемы.

6. *Таблицы.* Таблицы создаются при помощи элемента <table>. Таблицы могут иметь границы и состоять из произвольного количества строк и столбцов.

   Атрибут border = число создает прямоугольник с толщиной линии, равной указанному *числу*, который служит границей таблицы. Каждая строка в таблице начинается тегом <tr>, а тег <td> указывает на начало очередного элемента в данной строке. Ячейку с заголовком образует элемент <th> заголовок </th>, а подпись к таблице формируется элементом <caption> текст </caption>.

Апплеты

HTML является пассивным языком. Браузер отображает web-страницы в формате HTML и использует встроенные в страницу URL-адреса для перехода к другим страницам HTML, отображая текстовую информацию или рисунки. Уже сам по себе этот механизм предоставляет пользователям очень ценные возможности в отношении поиска информации в различных источниках. Результатом применения HTML явилось значительное изменение стереотипов поведения людей, которые заняты поисками каких-либо товаров, или участвуют в научных исследованиях, или ищут информацию какого-либо другого характера.

Но web-серверы обладают дополнительными возможностями, которые позволяют им играть более активную роль. Они могут получать информацию от клиентского браузера (то есть от пользователя) и соответствующим образом менять свое

поведение. Мы, в сущности, используем web-сервер для выполнения программ, и программирование web-страниц стало одним из последних изменений нашей парадигмы программирования. Такие выполняемые web-страницы стали известны, как апплеты, или мини-приложения. Некоторые из них доступны на всех web-серверах. Компания Sun Microsystems разработала язык Java, предназначенный для выполнения апплетов. Апплеты мы обсудим в разделе 12.2.3.

Формы. Форма представляет собой метод передачи информации между пользователем и web-сервером через браузер. Информация вводится пользователем на основе встроенных в HTML-файл команд. Затем эта информация передается некоторой программе на сервере. Это так называемый файл *общего шлюзового интерфейса* (Common Gateway Interface, CGI). На web-сервере обычно имеется специальный каталог cgi-bin, содержащий эти CGI-программы. Синтаксис элемента <form> задается как

```
<form method = "тип" action ="выполняемый cgi-сценарий"> текст </form>
```

где тип может быть или GET, или POST. При использовании метода GET данные передаются CGI-сценарию через переменную окружения QUERY_STRING, а в случае, если используется метод POST, информация передается через обычный процесс чтения. Сценарий CGI может быть программой на C или на Java, но чаще это shell-сценарий, или программа на языке TCL (Tool Command Language), или сценарий Perl. Мы уже обсуждали Perl как язык сценариев создания процессов с большими возможностями для обработки регулярных выражений (см. главу 3). Perl оказался также подходящим языком для обработки данных форм HTML. Мы приводим краткое описание сценариев Perl в разделе 12.2.2.

Существует несколько элементов, которые позволяют передавать информацию программе CGI.

Элемент <input type = text name="имя" size = число> в большинстве браузеров отображает на странице HTML поле ввода шириной в число символов, заданных в атрибуте size. Пара (*имя — введенный пользователем в поле ввода текст*) передается программе CGI, когда пользователь *отправляет* эти данные на сервер. Эта пара может быть организована в Perl как ассоциативный массив элементов данных для ассоциирования указанного имени и введенного пользователем текста.

Тег <input type=submit value = "Отправить"> создает кнопку с надписью Отправить, щелчок мышью на которой отправляет введенные в форме данные программе CGI на сервере. Тег <input type = "reset" value = "сброс"> создает кнопку с надписью сброс, щелчок на которой вызывает очистку всех введенных в форме данных и позволяет ввести новые данные до отправки формы на сервер.

С помощью тега <input>, задавая соответствующие значения атрибута type, можно создавать в форме другие элементы управления. Тег <input type = password> создает также текстовое поле для ввода пользователем информации с одним отличием: вводимый текст не отображается на экране (по умолчанию вместо любого символа отображается символ «звездочка» (*). Как следует из его названия, он используется для ввода пароля. Тег <input type = checkbox> создает элемент управления флажок, который может быть выбран или сброшен независимо от других флажков в форме. С помощью тегов <input type = radio> можно создать группу переключателей, из которых может быть выбран только один. Для этого следует при создании переключателей группы задать для них одно и то же значение атри-

бута name. Тег <textarea name = "name" rows = "num1" cols = "num2">, как и тег <input>, создает текстовое поле ввода, но с той разницей, что создаваемое поле может быть многострочным (значение атрибута rows больше единицы). Атрибут cols задает ширину текстового поля в символах. Использование форм, сценариев CGI и динамического построения файлов HTML позволяет WWW функционировать так же, как функционирует любая другая компьютерная среда. С помощью описанных средств можно разрабатывать поисковые системы, игровые программы и другие приложения, которые значительно повышают популярность WWW.

12.2.2. Сценарии CGI

На рис. 12.6 показано взаимодействие между web-страницей и сценарием CGI, написанным на языке Perl. Это простой пример клиент-серверного взаимодействия. Ввод URL в адресной строке web-браузера (стрелка *1* на рис 12.6) инициирует пересылку соответствующего файла HTML с сервера на локальный компьютер, где браузер отображает полученный файл на экране (стрелка *2* на рис. 12.6). После того как пользователь ввел необходимую информацию в поля формы полученной страницы HTML, кнопка Отправить используется для отправки этой информации на сервер, где она обрабатывается сценарием xaction, написанным на языке Perl (стрелка *3*). Сценарий Perl обрабатывает полученные данные и создает HTML-страницу, которая отображается в окне браузера пользователя. Эта страница (в нашем примере) содержит информацию, которая была передана на сервер (стрелка *4*).

Фактический код, который осуществляет это взаимодействие, приведен в листинге 12.2. Взаимодействие между web-страницей, написанной на HTML, и сценарием CGI определяется элементом <form>, приведенным в этом листинге. Последовательность действий, определяемая в этом листинге, такова.

1. Атрибут action элемента <form> указывает, что CGI-программа xaction будет активизирована при передаче данных формы на сервер.
2. Текст, введенный пользователем в первое поле ввода формы, будет ассоциирован с именем xfirst первого элемента <input>, а текст, введенный во второе поле, — с именем xlast второго элемента <input>. xfirst и xlast — произвольные имена, выбранные web-дизайнером, который создавал данную страницу.
3. Когда пользователь щелкает мышью на кнопке Отправить (в предположении, что были введены имена Marvin и Zelkowitz в первое и второе поля соответственно), на сервере вызывается программа xaction, которая читает переменную окружения QUERY_STRING со значением xfirst=Marvin&xlast=Zelkowitz.

На сервере Perl-сценарий xaction действует следующим образом:

1. Сначала сценарий информирует операционную систему о том, что он является программой Perl (первая строка в листинге 12.2, *б*).
2. Результат работы этой программы посылается обратно web-браузеру. Мы хотим, чтобы это была страница HTML, которую web-браузер сможет отобразить. Выводимая сценарием строка Content-Type : text/html сообщает web-браузеру, какие данные будут передаваться далее, — в данном случае это web-страница с текстом в формате HTML.

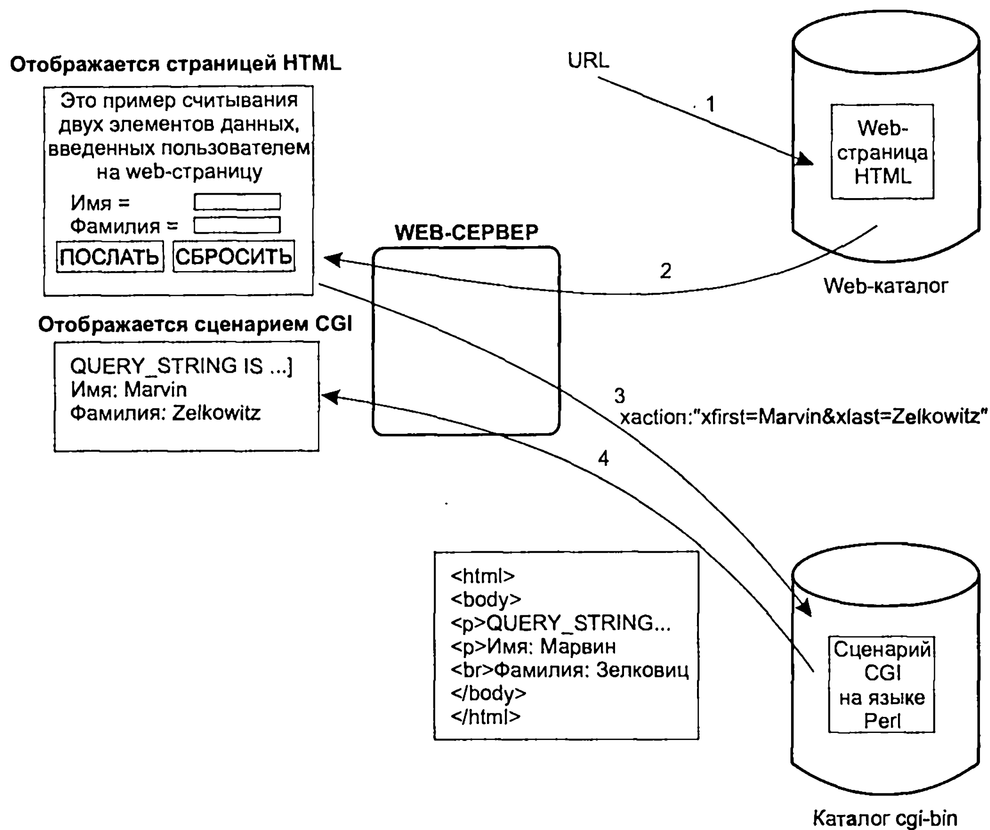


Рис. 12.6. Сценарии CGI

3. Каждый последующий оператор `print` языка Perl генерирует отображение очередной части страницы HTML. Программа таким образом динамически создает web-страницу, написанную на HTML, которую и будет отображать web-браузер.

4. Существенной частью алгоритма программы `xaction` является цикл `foreach`. Первая функция `split` разбивает содержимое параметра `QUERY_STRING` среды окружения операционной системы (хэш-массив `%ENV` в программе Perl), созданного элементом HTML `<form>`, на последовательность строк, используя в качестве разделителя символ `&`. Тем самым создается массив с двумя элементами `(xfirst = Marvin, xlast = Zelkowitz)`. Оператор `foreach` и функция `split` могут работать с произвольным количеством имен.

5. Вторая функция `split` разбивает каждый элемент созданного первой функцией `split` массива на два значения, используя в качестве разделителя символ `=`, которые присваиваются соответственно переменным `$key` и `$val`. Таким образом создаются пары значений (`xfirst`, `Marvin`) и (`xlast`, `Zelkowitz`), которые затем сохраняются в ассоциативном массиве `%tmp`, первое значение пары используется в качестве ключа, а второе — в качестве его значения. В дальнейшем доступ к сохраненным переданным пользователем дан-

ным может быть осуществлен с помощью конструкций `$tmp{'xfirst'}` и `$tmp{'xlast'}`.

Листинг 12.2. HTML-страница и сценарий Perl

а) web-страница

```
<HTML>
<HEAD>
<TITLE>тест cgi-сценария</TITLE>
</HEAD>
<BODY>
<p> Это пример считывания двух элементов данных,
введенных пользователем на web-страницу:
<form action="cgi-bin/xaction" method=get>
<p>Имя = <input type=text name=xfirst size=10>
<br>Фамилия = <input type=text name=xlast size=20>
<br> <input type=submit value=Послать>
<input type=reset value=Сбросить>
</form>
</BODY>
</HTML>
```

б) CGI-сценарий `xaction`, написанный на языке Perl

```
#!/usr/bin/perl
print "Content-Type: text/html\n\n";
print "<html><head>\n";
print "<title>Пример PERL-сценария</title>\n";
print "</head><body>\n";
print "<p>QUERY_STRING: $ENV{'QUERY_STRING'}\n";
foreach ( split( /&/, $ENV{'QUERY_STRING'}))
      { ($key, $val ) = split( /=/, $_, 2);
      $tmp{$key} = $val; }
print "<p> Имя: <b>$tmp{'xfirst'}</b>\n";
print "<p> Фамилия: <b>$tmp{'xlast'}</b>\n";
print "</body></html>\n"
```

Этот пример показывает только основной механизм взаимодействия между web-страницами и CGI-сценариями. Существуют и другие аспекты:

1. Однажды переданные и декодированные в операторе `foreach` параметры формы в дальнейшем могут быть использованы программой на Perl, например, для получения информации из файлов, расположенных на сервере, для оформления заказа пользователя Web на какие-либо товары или для выполнения какой-либо иной задачи.

2. Представленная в листинге 12.2, *б* операция выделения из переменной окружения QUERY_STRING переданных данных формы (вторая функция `split`) определена не полностью. Дело в том, что при передаче на сервер данных формы в строке запроса (метод GET) все пробелы во введенных пользователем в поля формы данных заменяются на символы +[1]. Таким образом,

[1] Пробелы могут заменяться не только на символ +, но и на шестнадцатеричное значение пробела %20. Замена пробелов, символов концов строк и других специальных символов в строке запроса на их шестнадцатеричные значения, начинающиеся с символа %, называется URL-кодированием. — *Примеч. науч. ред.*

все символы + следует заменить на пробелы с помощью следующей операции подстановки языка Perl:

```
$val=~s/+/ /g;
```

которая осуществляет глобальную подстановку обычного пробела во всей строке вместо символа +, на что указывает флаг g в конце операции подстановки.

3. Специальные символы, переданные в виде своих шестнадцатеричных значений %xx, должны быть преобразованы в соответствующие коды ASCII-символов по шестнадцатеричному коду xx:

```
$val=~s/%([0-9a-f]{2})/pack("c",hex($1))/gie;
```

Последняя команда — это подстановка, которая отыскивает символ % и следующие за ним два символа из множества десятичных цифр от 0 до 9 и букв от a до f (то есть из шестнадцати символов, которые используются для обозначения чисел в шестнадцатеричной системе) и упаковывает их в один символ. Таким образом, две шестнадцатеричные цифры, каждая из которых занимает четыре бита, в результате объединяются в один специальный символ, представимый восемью битами. Затем этот символ заменяет %xx в исходной строке.

12.2.3. Апплеты Java

Существующая на данный момент структура WWW с документами HTML, которые читаются web-серверами и отображаются web-браузерами, имеет четыре ограничения, хотя несомненным фактом является то, что возможности Web значительно превосходят первоначальные цели, поставленные физиками, работавшими в CERN.

1. При любом изменении протокола или добавлении новых протоколов все браузеры подлежат обновлению. Простое обновление сервера в соответствии с новым форматом является недостаточным, поскольку отображение документа зависит также от браузера, расположенного на клиентской машине.
2. Количество информации, доступное для сервера в HTML-процессе, ограничено данными, задаваемыми пользователями в элементах управления формы, создаваемой тегом <form> в документе HTML. Было бы полезно, чтобы HTML-программы (например, сценарии CGI) имели доступ к большим файлам данных на локальной системе браузера[1].
3. Управление взаимодействием сейчас осуществляется на сервере с помощью встроенных в web-страницы команд HTML. Это может сильно замедлить работу сервера, особенно если отображаемые страницы достаточно популярны и посещаются многими пользователями одновременно. Более эффективно было бы передать управление клиентскому браузеру, при этом роль сервера была бы скорее вспомогательной.

---

[1] Это высказывание авторов не соответствует действительности, так как в HTML предусмотрен элемент управления формы для передачи на сервер больших файлов <input type=file>, поддерживаемый практически всеми популярными браузерами. — *Примеч. науч. ред.*

4. Скорость отображения web-страниц ограничена скоростью передачи по линиям связи. Обычно клиентская машина большую часть времени находится в режиме ожидания информации, пока происходит загрузка данных. Типичная скорость этой загрузки – 33,6 кбит/с; иногда она бывает несколько выше – 56 кбит/с. Но с увеличением количества видео, аудио или графических компонентов web-страниц, имеющих большие размеры, задержки во времени при загрузке значительно увеличиваются.

Одним из способов решения этих проблем могло бы быть выполнение браузером некоторой части работы, которая традиционно выполняется на сервере. Вместо передачи данных HTML на сервер, сервер мог бы переслать браузеру небольшую программу и выполнять это приложение прямо на клиентской машине. Это позволило бы несколько разгрузить сервер и уменьшить количество информации, передаваемой с сервера на браузер.

Java – это язык программирования, синтаксис которого похож на синтаксис C++, а семантика – на семантику C++ и Smalltalk. Программы на Java первоначально компилируются в машинно-независимые интерпретируемые коды для виртуальной машины Java, для чего используется нотация, называемая *байт-коды*. Язык Java был разработан как средство выполнения программ в клиентском браузере.

При таком подходе безопасность браузеров является ключевым вопросом. Пользователь должен быть уверен, что браузер не загрузит какую-либо вредную программу с другого компьютера, которая затем может разрушить всю файловую систему на клиентском компьютере. Виртуальная машина Java снабжена механизмом проверки, которая осуществляется непрерывно во время ее работы и гарантирует безопасность загружаемых программ. Но этот вопрос, на самом деле, выходит за рамки рассмотрения языка Java и относится к вопросам безопасности систем.

Вначале компанией Sun был разработан web-браузер, названный HotJava, который включал в себя интерпретатор Java. Но с 1995 г. программы на Java обрабатываются практически всеми web-браузерами. Язык Java добавил новые возможности для работы в сети. Если раньше браузеры должны были «знать» обо всех протоколах и файловых форматах (например, http, ftp, gif и au), то браузер Java не нуждается в этом. При активизации гиперссылки проверяется ассоциированный с ней метод: если он недоступен, то браузер будет искать его сначала в своей локальной системе, а затем в системе сервера. Если он найден, то сервер перешлет его (при необходимости) браузеру на клиентскую машину, где он будет выполнен как локальный апплет.

Например, если HotJava обнаруживает http-гиперссылку, то сначала будет переслан с сервера соответствующий http-апплет, написанный на Java, а затем браузер HotJava выполнит его. Только те серверы, которые реализуют метод, ассоциированный с гиперссылкой, должны иметь апплет Java. Браузер же ничего не должен знать об этом апплете – он должен только иметь возможность принять его при необходимости.

Это позволяет развивать возможности сети несколько более упорядоченным образом. Если какая-либо компания разработала новый протокол, то она может реализовать его на своем сервере, снабдив последний апплетом Java для обработки этого нового метода. Когда браузер встречает этот новый метод, он может загрузить соответствующий апплет и затем выполнить его. Одна из особенностей

такого подхода (или одна из опасностей, с другой точки зрения) — это то, что компания может сделать эту услугу платной и превратить в источник дохода, хотя в настоящее время такая ситуация встречается довольно редко. Разумеется, некоторые считают, что доступ к сети должен быть бесплатным. Но, с нашей точки зрения, бесплатный доступ не может существовать вечно — необходимы какие-то разумные схемы оплаты. Но здесь может скрываться некий подвох — компания может использовать уже существующий протокол, но просто изменить его название и взимать плату, как если бы он был новым и требовал дополнительных затрат. Мы надеемся, что такого не произойдет — иначе вся система Интернета может рухнуть. Развитие и популярность Интернета основаны на универсальности доступа к Интернет-ресурсам.

Побочный эффект использования апплетов заключается в том, что, поскольку он выполняется на клиентской машине, у него появляется возможность доступа к файловой системе пользователя. Виртуальная машина Java, в принципе, гарантирует, что этого не произойдет. Разумеется, безопасность файловой системы пользователя — одно из важнейших требований. При разработке браузеров следует быть крайне внимательными и осторожными, чтобы браузер, получивший доступ к апплету, мог распознать присутствие в нем вируса (вирус изменяет выполняемые программы на машине пользователя таким образом, что могут разрушаться файлы) или «троянского коня» (программы, которая перехватывает управление пользовательским компьютером) и не допустить выполнение такого апплета на клиентском компьютере.

Активные апплеты. Использование сценариев CGI позволяет разработчику web-страницы организовать передачу информации в обоих направлениях между web-сервером и пользователем, использующим web-браузер. Тем не менее область применения CGI-сценариев весьма ограничена, так как ограничен объем информации, который можно передать с помощью элемента HTML `<form>`. Апплеты Java позволяют передавать дополнительную информацию, так как Java — это полноценный язык программирования и его возможности не ограничены простым заполнением полей в элементе HTML `<form>`.

Для обработки исходных программ Java в HTML был добавлен элемент `<applet>`:

```
<applet code = апплет_Java width=num1 height=num2>
```

Результатом обработки этого элемента будет выполнение откомпилированного в промежуточный байт-код апплета Java — выполняемой Java-программы, находящейся в том же каталоге, что и HTML-страница. Этот апплет будет отображать свои результаты в окне шириной num_1 пикселов и высотой num_2 пикселов.

Например, для выполнения традиционной программы `Hello, World!` можно написать программу на Java и сохранить ее в файле hello.java:

```
import java.awt.*; /* библиотека апплетов */
public class hello extends java.applet.Applet {
        public void paint(Graphics x) {
            x.drawString("Hello, World!", 100, 100);
        }
}
```

При компиляции этой программы компилятором Java создается файл hello.class. Теперь этот файл может быть выполнен при отображении web-страницы в браузе-

ре, поэтому он помещается в соответствующий каталог, откуда web-браузер может его прочитать.

Для инициализации этого апплета используется следующий код HTML:

```
<html>
<body>
<applet code = "hello.class" width = 200 height = 200>
</applet>
</html>
```

Если URL-адрес этой web-страницы передается web-браузеру, элемент <applet> инициирует выполнение Java-программы hello.class. В этом случае Java использует метод paint апплета для отображения вывода из программы на web-страницу (подобно тому, как действует команда showpage языка Postscript). В этом небольшом апплете метод paint переопределяет метод paint, определенный по умолчанию в классе Applet из пакета java.applet, и отображает надпись Hello, World!.

12.2.4. XML

HTML как приложение SGML имеет фиксированный набор семантических правил для каждого элемента. Как было сказано ранее, существуют конфликты между семантикой содержимого документа (например, элементами заголовков, абзацев, списков) и результатами представления (например, шрифтами, цветами, высотой и шириной объектов). Для устранения этого несоответствия в конце 90-х гг. Консорциум WWW (W3C) разработал расширяемый язык разметки (Extensible Markup Language, XML) как возможную замену HTML для отображения web-страниц.

XML – это еще один экземпляр SGML. Документ XML состоит из множества элементов, использующих синтаксис SGML <element имя>. Как и в случае с документами SGML, объявление типа документа (DTD) содержит описание семантики каждого типа элементов. Тем не менее, как и в случае с HTML, некоторые из элементов становятся стандартными и многие документы XML могут обрабатываться без наличия внешнего DTD.

Документ XML начинается с <?xml XMLVersionNumber ?> и состоит из вложенных разделов элементов вида <имя> текст </имя>. Элементы без завершающих тегов задаются в виде <имя/>.

Многие концепции HTML были перенесены в XML. URL-адреса (универсальные указатели ресурса) HTML превратились в XML в *универсальные идентификаторы ресурса* (Uniform Resource Identifier, URI), которые определяются в XML как *простые ссылки* (simple links):

```
<a xml:link="simple" href="обычный http URL">
```

Элементы состоят из разметки и собственно содержимого. Существует шесть типов разметки.

1. *Элементы.* Этот тип знаком нам по HTML и определяется как <name> текст </name>.
2. *Атрибуты.* Атрибуты располагаются в начальном теге элемента. В приведенном выше примере с URL атрибутом является href="обычный http URL".

3. *Ссылки на сущности.* Символы, подобные <, имеют специальное значение в XML. Поэтому, когда в документе требуется такой символ, используемый в своем обычном значении, он задается с помощью ссылки на сущность, в данном случае на <:. Например, <:start> равносильно <start>. Для некоторых других символов также имеются ссылки на сущности.
4. *Комментарии.* Комментарии задаются следующим образом: <!-- текст -->.
5. *Инструкции по обработке.* Эти инструкции задаются как <?приложение-получатель инструкция?>. Приложение-получатель должно выполнять те инструкции, которые оно распознает, и игнорировать остальные. Инструкции по обработке являются механизмом для передачи специфической информации о протоколах приложению, которое обрабатывает XML-страницу.
6. *CDATA.* Это символы, которые игнорируются в процессе разметки и передаются непосредственно приложению. Например, текст в <![CDATA[текст]]> передается непосредственно приложению.

Основное различие между XML и HTML заключается в том, что для документов XML с помощью элемента <!DOCTYPE...> может быть задано определение типа документа (DTD). Это позволяет задать синтаксис и семантику для элементов документа XML, которые не являются просто значениями по умолчанию для соответствующих тегов, как в случае HTML. Например, web-страница, написанная на HTML, может быть определена в XML с использованием DTD:

```
<!DOCTYPE HTML PUBLIC "URI документа со спецификациями HTML" >
```

В каждой области деятельности (например, в банковских электронных системах, в электронной коммерции, в библиотечных базах данных) можно определить DTD, которое задает семантику той информации, которую необходимо обрабатывать. Это позволит обрабатывать специфические для данной области web-страницы XML эффективнее, чем при использовании более простого формата HTML.

12.3. Рекомендуемая литература

Подробное описание Postscript можно найти в *Справочном руководстве по языку Postscript* [7]. Об использовании TEXрассказывает автор этого языка [66], а расширение LATEX, разработанное Лампортом, описано в [69]. Более полное описание сетевых технологий вы найдете в [34], а дополнительную информацию по web-разработкам — в [16].

12.4. Задачи и упражнения

1. Сколько проходов потребуется LATEXдля создания этой книги? Следует учитывать, что в этой книге имеется оглавление, главы с заголовками различных уровней и предметный указатель. Какие данные создаются при каждом проходе? Какое максимальное количество проходов необходимо?
2. Напишите программу на Postscript, которая печатает объявление, рекламирующее курс лекций, в котором используется эта книга. Ваше объявле-

ние должно содержать несколько строчек текста и как минимум пять рисунков.

3. Предложите несколько способов сжатия файла Postscript для его передачи с использованием меньшего количества байтов.
4. Опишите несколько примеров смешивания в HTML семантического содержимого web-страницы и ее представления на экране дисплея.
5. Каковы преимущества и недостатки использования языка Postscript вместо HTML как языка отображения web-страниц?
6. Почему Perl хорошо подходит для создания сценариев CGI? Почему выбор C для этой цели неудачен?

Приложение. Обзоры языков

В этой книге мы акцентировали внимание на синтаксисе и семантике тринадцати различных нотаций: Ada, C, C++, FORTRAN, HTML, Java, LISP, ML, Pascal, Perl, Postscript, Prolog и Smalltalk. Язык HTML был достаточно хорошо описан в главе 12. В данном приложении мы представляем примеры программ на остальных двенадцати языках, которые менее подробно рассматривались в этой книге, а также приводим основы синтаксиса и семантики, необходимые для написания простых программ на каждом из этих языков.

П.1. Ada

Пример с пояснениями

В листинге П.1 приведен пример программы суммирования элементов массива. Здесь мы используем инкапсуляцию, обеспечиваемую пакетами языка Ada, чтобы реализовать сокрытие информации. Для входных данных

```
4 1 2 3 4 5 1 2 3 4 5 0
```

программа печатает следующие результаты:

```
  1 2 3 4
SUM = 10
  1 2 3 4 5
SUM = 15
```

Листинг П.1. Пример суммирования элементов массива на языке Ada

```
 1 package ArrayCalc is
 2     type Mydata is private:
 3     function sum return integer;
 4     procedure setval(arg:in integer);
 5     private
 6         size: constant:= 99;
 7         type myarray is array(1..size) of integer;
 8         type Mydata is record
 9             val: myarray;
10             sz: integer := 0;
11             end record;
12         v: Mydata;
13 end;
14 package body ArrayCalc is
```

```
15    function sum return integer is
16       temp: integer:
17    -- Тело функции sum
18       begin
19       temp := 0:
20       for i in 1..v.sz loop
21          temp := temp + v.val(i):
22          end loop:
23       v.sz := 0:
24       return temp:
25       end sum:
26    procedure setval(arg:in integer) is
27       begin
28       v.sz := v.sz+1:
29       v.val(v.sz) := arg:
30       end setval: end:
31 with Text_IO: use Text_IO:
32 with ArrayCalc: use ArrayCalc:
33 procedure main is
34    k. m: integer:
35 begin -- начало главной программы main
36    get(k):
37    while k>0 loop
38       for j in 1..k loop
39          get(m): put(m. 3):
40          setval(m):
41          end loop:
42       new_line: put("SUM ="):
43       put(ArrayCalc.sum.4):
44       new_line: get(k):
45       end loop:
46 end:
```

Строка 1. Спецификация пакета ArrayCalc, который собирает массив данных для этой программы. Все детали о пакете ArrayCalc, которые должны быть известны за пределами этого пакета, следует задать в этой спецификации.

Строка 2. Тип данных MyData будет известен за пределами данного пакета. Однако этот тип определен как private, а значит, можно объявлять объекты типа MyData и передавать их функциям, описанным только внутри пакета ArrayCalc.

Строки 3–4. Здесь определяются подпрограммы sum и setval. Указаны только сигнатуры двух подпрограмм, чтобы разрешить компиляцию вызовов этих подпрограмм из других пакетов.

Строка 5. Ключевое слово private обозначает, что вся последующая информация в этой спецификации пакета не может использоваться в подпрограммах, определенных за пределами пакета ArrayCalc.

Строка 6. Имя size определяется как константа, равная 99. Область видимости данной константы ограничивается пакетом ArrayCalc.

Строки 8–12. Описываются локальные данные пакета, а именно Mydata и v.

Строка 14. Здесь начинается реализация пакета ArrayCalc, где задаются детали реализации каждой подпрограммы, объявленной в спецификации пакета.

Строка 17. Комментарии в языке Ada начинаются с символов -- и продолжаются до конца строки.

Строки 18–25. Блок begin является телом функции sum.

Строки 20–22. Итерационный процесс управляется оператором loop. Существует несколько его вариантов. В данной подпрограмме используется вариант цикла for, который повторно выполняет тело цикла, пока переменная цикла i принимает последовательные значения от 1 до v.sz. Заметим, что переменная i не была объявлена, но она неявно объявляется как локальная переменная, действующая внутри оператора цикла.

Строки 26–30. Здесь определяется подпрограмма setval. Формальный параметр arg является только входным и аналогичен параметру в языке Pascal, передаваемому по значению.

Строка 31. Ключевое слово with указывает, что в следующем далее компоненте языка Ada должен использоваться стандартный пакет Text_IO. Все объекты, определенные в Text_IO, в качестве префикса содержат имя пакета. Поскольку Text_IO является стандартным пакетом ввода-вывода, среди прочих в нем определены функции Text_IO.get и Text_IO.put.

Команда use специфицирует, чтобы компилятор Ada первым просматривал пакет Text_IO для определения области видимости объектов. Это позволяет в программе использовать, например, имя put вместо полного имени Text_IO.put.

Строка 32. Пакет ArrayCalc также будет использоваться в следующем далее компоненте Ada.

Строка 33. Программа, написанная на языке Ada, состоит из главной процедуры, которая может вызывать подпрограммы из различных пакетов. В этом примере главная процедура называется main.

Строка 36. Для чтения данных в пакете Text_IO определена функция get. Это перегруженная функция, так что аргумент может быть целой переменной (в данном примере k), строковой переменной или парой (файл, объект), позволяющей считывать данные из файлов. Поскольку для каждого случая набор аргументов уникален, то компилятор сам знает, какую функцию get следует вызывать.

Строка 39. Функция get считывает данные в целую переменную m. Функция put, определенная в пакете Text_IO, выводит на печать значение своего аргумента. Необязательный второй аргумент задает ширину поля вывода для печати значения объекта. Так, значение переменной m печатается в виде трех символов (с предшествующими пробелами для чисел, меньших 100).

Строка 40. Вызывается процедура setval пакета ArrayCalc, которой передается фактический параметр m. Поскольку в строке 32 имеется команда use, то нет необходимости в задании полного имени процедуры ArrayCalc.setval.

Строка 42. Функция new_line, определенная в пакете Text_IO, выводит на печать символ конца строки. Функция put печатает SUM =. Обратите внимание на то, что этот вызов функции отличается от вызова функции put в строке 39, поскольку данная функция печатает строку, а функция put в строке 39 печатает целое число. Реализация перегрузки функций определяет, какую функцию следует вызвать.

Строка 43. Здесь употребляется полностью уточненное имя. Из-за того, что в строке 32 использовалась команда use, вместо этого имени можно было бы написать просто sum. Полностью уточненное имя необходимо использовать, если одно и то же имя функции используется в нескольких пакетах внутри области видимости текущего оператора и при разрешении перегрузки они окажутся неразличимы. В данном случае число, возвращаемое функцией sum, печатается в виде четырех выходных символов.

П.1.1. Объекты данных

Основные типы данных, описанные в этом разделе, лишь слегка расширяют типы данных, доступные в языке Pascal. Однако средства Ada, позволяющие программисту определять новые типы, значительно мощнее, чем аналогичные в языке Pascal, а наличие пакетов позволяет инкапсулировать определения типов для получения истинных абстракций данных.

Элементарные типы данных

Такие типы, как целый (*integer*), вещественный (*real*) (в языке Ada называемый *float*), символьный (*character*), булев (*Boolean*) и строковый (*string*) являются *предопределенными* типами, определенными в пакете Standard, который автоматически известен внутри любой программы на языке Ada. Фактически эти типы определены при помощи набора более примитивных *конструкторов типов*. Примерами таких конструкторов могут служить *перечисления*, *массивы* и *записи*. Например, типы Boolean и character определяются как перечисления, а строковый тип string определяется как вектор, состоящий из символов. Поскольку механизм определения типов используется как для элементарных, так и для определенных пользователем типов, многие примеры этого раздела могут включать определения типов.

Переменные и константы. Любой объект данных может быть определен либо как переменная, либо как константа. Любое объявление, начинающееся с ключевого слова constant, является объявлением константы. В нем должно быть задано значение константы, которое не может меняться в процессе выполнения программы. Если слово constant опущено, тогда это же объявление определяет объект данных как переменную. В этом случае можно задать начальное значение, а затем изменять его обычным присваиванием. Например, константу MaxSize и переменную CurrentSize можно объвить следующим образом:

```
MaxSize: constant integer := 500;
CurrentSize: integer := 0;
```

Константы языка Ada могут быть как объектами элементарного типа, так и массивами или записями:

```
WeekDays: constant array(1..5) of string(1..3) :=
    ("MON", "TUE", "WED", "THU", "FRI")
```

В языке Ada существует множество предопределенных *атрибутов*, которые обозначают важные связи или свойства объектов данных, типов данных, подпрограмм или аппаратной части компьютера. Для получения значения атрибута следует указать его имя (предварив его префиксом «'») сразу же после имени объекта. Например, одним из атрибутов любого векторного типа данных, Vect, является нижняя граница диапазона изменения его индекса, обозначаемая как Vect'First. Верхняя граница обозначается Vect'Last. Используя этот атрибут, можно написать программу так, что она будет независима от конкретного определения типа Vect. Таким образом, если изменяется определение Vect, операторы, использующие Vect'First и Vect'Last, можно не изменять. Предопределенные атрибуты существуют для большинства основных типов данных, описанных ниже; другие атрибуты обсуждаются в следующих разделах.

Числовые типы данных. Основными числовыми типами данных являются целые числа, вещественные числа с плавающей точкой и вещественные числа с фик-

сированной точкой. Объявления объектов этих типов похожи на объявления в Pascal, где используются атрибуты range (для целых чисел) и digits (для вещественных чисел), которые определяют диапазон значений для объектов:

`type DayOfYear is range 1..366;`	Целые значения от 1 до 366
`MyBirthday: DayOfYear := 219;`	MyBirthday соответствует дате 7 августа
`type Result is digits 7;`	Вещественный тип из 7 цифр
`Answer: Result;`	Вещественная переменная из 7 цифр

Программа может получить доступ к аппаратному представлению вещественного числа с фиксированной точкой, если таковое существует. Для определения типа с фиксированной точкой в программе следует объявить необходимую максимальную разность между двумя соседними значениями объектов этого типа, называемую *дельта-значение*. Например, для определения десятичных чисел с фиксированной точкой с двумя знаками после десятичной точки дельта-значение следует задать как .01:

```
type SmallAmount is delta 0.01 range -100.0..100.0;
```

Для чисел с фиксированной точкой предусмотрены обычные арифметические операции. Если в аппаратной части непосредственно не предусмотрено соответствующее представление с фиксированной точкой, в реализации языка может использоваться моделируемое программными средствами представление числа с плавающей точкой, если обеспечена определенная точность представления числа (то есть числа должны быть представлены с точностью не меньшей, чем дельта-значение).

Перечисления. В языке Ada перечисления можно определять подобно тому, как они определяются и реализуются в Pascal. Например, определение

```
type class is (Fresh, Soph, Junior, Senior);
```

задает перечисляемый тип, который затем можно использовать при объявлении переменных. В отличие от языка Pascal перечисляемый тип не может быть задан непосредственно при объявлении переменной, а должен быть определен как отдельное описание типа. Для представления элементов перечисления во время выполнения программы используется номер позиции для каждого литерального значения: 0 для первого из перечисленных значений, 1 — для второго и т. д. Для перечисляемых значений предусмотрены основные операции сравнения (например, равенство, меньше чем и т. д.), а также функции определения последующего и предыдущего элементов и операция присваивания.

Одно и то же литеральное имя может использоваться в нескольких перечислениях (в отличие от языка Pascal, где все перечисляемые литеры должны быть различными). Говорят, что такое литеральное имя является *перегруженным*. Например, если задано предыдущее определение class, то определение

```
type BreadStatus is (Stale, Fresh);
```

перегружает литеральное имя Fresh. В некоторых случаях компилятор может разрешить перегрузку непосредственно (например, при присваивании B := Fresh, где B — объект типа class). Для явного определения того, какое из значений Fresh имеется в виду в данном месте программы, где видны оба типа (class и BreadStatus), программист может указать вместе с литеральным именем имя базового типа (например, class(Fresh) или BreadStatus(Fresh)).

Символьный и булев типы. Эти типы определены в пакете Standard как специальные перечисления.

Указатели. Тип указатель, называемый ссылочным типом (access), вместе с элементарной функцией new создает новый объект данных (выделяя для него память в куче) и возвращает указатель на него, который затем может быть присвоен переменной ссылочного типа. Переменную нельзя непосредственно объявить как переменную ссылочного типа, а следует использовать следующее описание типа

```
type access_typename is access имя_типа;
```

После этого переменные можно объявлять с помощью определенного таким образом ссылочного типа. Переменные ссылочного типа, следовательно, могут указывать на объекты данных только одного типа. Все переменные ссылочного типа имеют по умолчанию в качестве начального значения указатель на несуществующий объект, NULL, если только при их объявлении им явно не присваивается указатель на какой-либо объект.

Для размещения всех объектов данных конкретного ссылочного типа можно зарезервировать область памяти, используя следующий оператор:

```
for имя_ссылочного_типа use выражение
```

Здесь выражение задаст необходимый для резервирования размер области памяти. Любое последующее использование оператора new для этого ссылочного типа выделяет память внутри отведенной области памяти.

Структурированные типы данных

Векторы и массивы. Объект данных массив может быть любой размерности и иметь произвольные диапазоны изменения индексов и компоненты любого типа. Например, следующее объявление:

```
Table: array (1..10.1..20) of float;
```

создает матрицу вещественных чисел размером 10 × 20. Для выбора компонентов используется операция индексации (например, Table(2. 3)).

Для объявления классов массивов можно использовать описание типов. В языке Ada устранен недостаток языка Pascal — границы массива не включаются в обозначение типа. При определении массива как типа в языке Ada можно не задавать конкретный диапазон индексов для одной или нескольких размерностей. Конкретная размерность задается при объявлении объекта данных, принадлежащего к этому типу. Например, определение типа *массив* может быть следующим:

```
type Matrix is
      array (integer range <>. integer range <>) of float;
```

где угловые скобки <> обозначают диапазон изменения индексов, который необходимо задать при объявлении переменной типа Matrix. Последующее объявление переменной задает конкретный диапазон индексов для объекта типа Matrix, например:

```
Table: Matrix(1..10.1..20);
```

Диапазон индексов не обязательно должен быть подмножеством целых чисел, он может быть объектом любого перечисляемого типа. Границы диапазонов изменения индексов также могут задаваться выражениями, которые содержат вычисленные результаты, например:

```
NewList: array (1..N) of ListItem;
```

где N — переменная с вычисленным значением. Каждый раз при входе в подпрограмму, содержащую данное объявление, в зависимости от значения переменной N в момент входа заново задается длина вектора NewList.

Индексацию можно использовать для выбора не только единичного элемента массива, но и *сечения* массива — непрерывной последовательности компонентов исходного вектора. Например, ссылка NewList(2..4) возвращает трехкомпонентное сечение, содержащее компоненты вектора NewList со второго по четвертый.

Инициализация. При объявлении массива его компонентам можно присваивать начальные значения:

```
PowerOfTwo: array (1..6) of integer := (2,4,8,16,32,64);
```

Символьные строки. Символьные строки трактуются как предопределенный векторный тип, в определении которого используются два других предопределенных типа — positive (положительные целые) и character (перечисление символов, определенное в пакете Standard):

```
type string is array (positive range <>) of character;
```

Таким образом, конкретный строковый объект можно объявить, задав максимальную длину (диапазон изменения индекса) строки:

```
MyString: string(1..30);
```

Файловый тип данных. Файлы и операции ввода-вывода в языке Ada определены как абстрактные типы данных, использующие некоторые стандартные пакеты. Кроме этих пакетов реализация, как правило, должна поддерживать дополнительные пакеты ввода-вывода, построенные с использованием основных типов данных и операций, определенных в стандартных пакетах (например, пакет графического вывода на экран).

Определяемые пользователем типы

В языке Ada объекты данных записи похожи на записи языка Pascal. Запись может иметь любое число компонентов произвольного типа. Если запись имеет несколько вариантов, вариантные компоненты должны располагаться в конце. Однако в отличие от языка Pascal запись с вариантами всегда должна иметь поле признака, называемое *дискриминантом* (тем самым устраняется еще один дефект языка Pascal), которое во время выполнения программы указывает, какой из вариантов записи используется в настоящее время. Попытки ссылки на компонент, который не содержится в текущем варианте записи, отслеживаются во время выполнения программы и воспринимаются как исключительные ситуации.

В отличие от языка Pascal в языке Ada для определения записи нужно, во-первых, задать *описание типа* для определения структуры записи и затем объявить переменную, использующую имя этого типа. Аналогично если компонент записи сам является записью, то эта запись должна быть определена с помощью отдельного определения типа, причем определение второй записи не может быть вложено внутрь определения большей записи.

Для любого компонента записи можно задать *начальное значение по умолчанию*:

```
type Birthday is
       record
          Month: string(1..3) := "Jan":
          Day: integer range 1..31 := 17:
          Year : integer range 1950..2050 := 1969:
       end record:
```

Каждый объект данных типа Birthday имеет начальное значение ("Jan", 17, 1969), если только в объявлении не присваивается другое начальное значение. Если объявить переменную MyDay следующим образом:

```
MyDay : Birthday:
```

то MyDay будет иметь начальные значения по умолчанию для своих компонентов. Однако если переменную YourDay объявить как

```
YourDay : Birthday := ("MAR", 13, 196B):
```

или

```
YourDay : Birthday := (Month => "MAR", Day => 13, Year => 1968):
```

то вместо начальных значений, присваиваемых по умолчанию, используются указанные значения.

Для определения похожих объектов данных объявления пользовательских типов могут содержать параметры, называемые *дискриминантами*. Например, для определения типа записи, которая содержит информацию о каждом месяце, включающем вещественный вектор, имеющий свой элемент для каждого дня месяца, можно использовать следующее определение:

```
type Month (Length : integer range 1..31) is
       record
          Name : string(1..3):
          Days : array (1..Length) of float:
       end record:
```

Затем можно объявить конкретные записи типа Month:

```
Jan : Month(31):
Feb : Month(28):
```

Для доступа к каждому компоненту вектора без знания явных границ каждого месяца можно использовать атрибуты Days'First и Days'Last.

Второй способ использования одного определения типа запись для определения похожих, но не одинаковых объектов этого типа заключается в использовании структуры вариантной записи, похожей на такую же структуру языка Pascal. Поле признака, используемое для указания нужного варианта, всегда перечисляется как дискриминант в определении типа записи (в той же форме, что представлена выше). Например, запись о служащем может быть представлена как

```
type WageType is (hourly, salaried):
type EmployeeRec (PayType : WageType) is
       record
          Name: string(1..30):
          Age: integer range 15..100:
          case PayType is
             when Hourly =>
                Hours: float:
                HourlyRate: float:
             when Salaried =>
```

```
            MonthlyRate: float;
    end record;
```

Затем можно объявить конкретные объекты записи:

```
President: EmployeeRec(Salaried);
Chauffeur: EmployeeRec(Hourly);
```

где выбранный вариант не изменяется во время выполнения программы (например, тип оплаты за работу President никогда не изменится на Hourly).

Если необходимо изменить вариант записи во время выполнения программы, следует осуществить присваивание переменной всей записи — никаким другим способом не изменить значение дискриминанта. В результате такого присваивания в блок памяти, представляющий запись, записывается полный набор новых значений, включая новое значение дискриминанта.

Подтипы. Любой определенный тип дает имя классу объектов, имеющих определенные атрибуты. *Подтип* этого *базового типа* состоит из подмножества тех объектов данных, которые удовлетворяют конкретным *ограничениям*. Ограничением для базового типа integer может быть сужение множества целых чисел до некоторого подмножества; ограничением на базовый тип array, в определении которого не указан диапазон изменения индексов (<>), может быть конкретный диапазон изменения индексов. Ограничением на базовый тип, определенный как запись с дискриминантом, может быть конкретное значение дискриминанта (которое может определять конкретный вариант записи).

Синтаксис определения подтипа следующий:

```
subtype имя_подтипа is имя_базового_типа ограничение;
```

как, например, в

```
subtype GraphScale is integer range -20..20;
subtype Executive is EmployeeRec(Salaried);
```

Подтип не определяет новый тип, он просто определяет подмножество объектов данных, принадлежащих базовому типу. В частности, для объектов подтипа можно использовать те же операции, которые используются для объектов базового типа, а объекты разных подтипов одного общего базового типа можно смешивать в качестве операндов одного выражения.

Производные типы. Производный тип похож на подтип, но он является *новым типом*, отличным от базового. Синтаксис определения производного типа следующий:

```
type имя_производного_типа is new имя_базового_типа;
```

где базовый тип можно определить с ограничениями (и, таким образом, в действительности это будет подтип). Например, можно записать:

```
type Meters is new integer;
type Liters is new integer;
```

Производный тип позволяет использовать все те же операции, что и базовый тип (или подтип). Однако, поскольку производный тип отличается от базового, операции не могут применяться к смеси операндов базового и производного типов. Например (возвращаясь к приведенным ранее определениям производных типов), поскольку операция + определена для базового типа integer, следовательно, она определена и для производных типов Meters и Liters.

Однако если X, Y и Z — переменные типа integer, Meters и Liters соответственно, то ни одну пару, составленную из этих переменных, нельзя между собой складывать (то есть запись X + Z некорректна). Таким образом, используя производные типы, программист защищается от случайного сложения двух переменных, представляющих различные виды данных, хотя основное представление данных может быть одинаковым (в данном примере все переменные представлены целыми числами).

Иерархия подтипов Ada

- Все типы
  - элементарные
    - скалярные
      - дискретные
        - перечисления
          - символьные
          - булевы
          - другие перечисления
        - целые числа
          - целые со знаком
          - целые без знака
      - вещественные
        - с плавающей точкой
        - с фиксированной точкой
          - обычные
          - десятичные
    - ссылки
      - ссылки на объекты
      - ссылки на подпрограммы
  - составные
    - массив
      - строка
      - другие массивы
    - недискриминантные записи
    - дискриминантные записи
    - задачи
    - защищенные

Числовые типы и неограниченные типы в эту иерархию не входят.

П.1.2. Управление последовательностью действий

Выполнение программ на языке Ada является операторно-ориентированным, и в этом язык Ada похож на FORTRAN и Pascal.

Выражения

Для обычных бинарных арифметических и логических операций и операций отношения используется инфиксная запись. Префиксная запись используется для унарных операций (+, - и not), а для вызовов функций используется обычное мате-

матическое префиксное представление имя_функции(список_аргументов). При отсутствии скобок приоритеты распределены в следующей последовательности (сначала перечислены операции с наивысшим приоритетом):

```
**  (возведение в степень)
*   /    mod   rem
+   -    abs   not (унарный + и - )
+   -    &  (бинарный + и - )
=   /=   <     <=     >     >=    in    not in
and or   xor   and then     or else
```

Операции с одинаковым приоритетом ассоциативны слева направо. Обратите внимание на то, что в языке Ada присутствуют булевы операции, вычисляемые по укороченной схеме, and then и or else, которые вычисляют только свой первый операнд, если его значения достаточно для вычисления значения всего выражения. Операции in и not in используются для проверки, принадлежит ли значение конкретному диапазону или удовлетворяет ли конкретным ограничениям.

Операторы

Операторные структуры, управляющие последовательностью действий, включают в себя обычные условные операторы (if и case) и операторы цикла (loop). Предусмотрено три формы оператора цикла, а оператор exit позволяет осуществить прямой выход из цикла. Также предоставляется ограниченная форма оператора goto. Все основные управляющие операторы заканчиваются ключевым словом end, за которым следует ключевое слово, определяющее оператор (например, end if, end loop), так что для объединения операторов в одну программную единицу редко требуется специальная пара begin-end.

Присваивание. Оператор присваивания языка Ada похож на оператор присваивания языка Pascal. Также возможно присваивание полного набора значений переменной типа запись, если использовать один из следующих способов:

```
MyDay := ("DEC".30.1964):
MyDay := (Year => 1964. Month => "DEC". Day => 30):
```

Оператор if. Оператор if имеет общую форму:

```
if булево_выражение then
      - последовательность операторов
elsif булево_выражение then
      - последовательность операторов
elsif булево_выражение then
...
else
      - последовательность операторов
end if;
```

в которой любая из частей elsif или else может быть опущена.

Оператор case. Общая форма оператора case имеет следующий вид:

```
case выражение is
      when вариант | ... | вариант => последовательность операторов;
      when вариант | ... | вариант => последовательность операторов;
...
when others => последовательность операторов;
end case:
```

Каждый вариант является либо одиночным значением, либо диапазоном значений, которые может принимать выражение, записанное в заголовке оператора. Таким образом, вариант — это литеральное значение или подмножество целых чисел (или значения другого перечисляемого типа), например:

```
case GradeLevel is
        when Fresh => операторы:
        when Soph | Junio => операторы:
        when Senior => операторы:
end case;
```

Должно выполняться следующее требование: либо варианты выбора охватывают весь диапазон возможных значений выражения, либо имеется завершающая конструкция when others для явного описания действий для всех не указанных значений выражения. Таким образом, в процессе выполнения программы исключается возможность возникновения ошибки, которая появляется, если для вычисленного значения выражения в операторе case не определено никаких действий. Если же для некоторых значений выражения не должно производиться никаких действий, соответствующая часть оператора case может содержать ключевое слово null.

Оператор цикла. Основной итерационный оператор имеет форму

```
loop
- последовательность операторов
end loop
```

Поскольку в заголовке оператора цикла не указаны условия его окончания, данная форма цикла будет выполняться бесконечно, пока он не будет прерван либо явным оператором exit, goto или return, либо по причине возникновения исключительной ситуации.

Контролируемый итерационный процесс может быть организован с помощью форм оператора цикла while или for. Оператор while имеет синтаксис

```
while булево_выражение
```

а оператор for бывает двух видов:

```
for имя_переменной in дискретный_диапазон
for имя_переменной in reverse дискретный_диапазон
```

где дискретный_диапазон может быть некоторым диапазоном перечисляемого или целого типов (например, 1..10, или Fresh..Junior). Ключевое слово reverse означает, что итерация начинается с конечного значения диапазона, присвоенного данной переменной, и продолжает работать, пока не достигнет начального значения. Примеры циклов for приведены в листинге П.1.

Важной особенностью цикла for является то, что имя переменной, записанное в заголовке цикла, *не* объявляется как обычная локальная переменная (как в языке Pascal). Вместо этого она *неявно* объявляется в момент ее появления в операторе for и ее можно использовать только внутри этого цикла. Как только цикл заканчивает свою работу, такая переменная исчезает, и ее больше нельзя использовать. Эта структура позволяет компилятору языка Ada осуществить несколько важных оптимизаций цикла.

Оператор exit. Оператор exit имеет следующий вид:

```
exit when булево_выражение;
```

или просто exit. При выполнении указанного в операторе exit булева условия управление передается в конец самого внутреннего цикла, включающего этот оператор. Альтернативная возможность заключается в использовании меток для операторов циклов, в этом случае оператор exit должен содержать метку цикла, из которого следует выйти, например:

```
exit OuterLoop when A(I,J) = 0;
```

Оператор goto. В языке Ada имеется оператор goto, но его использование строго ограничено. Вообще, оператор goto может передать управление только другому оператору внутри той же подпрограммы либо тому же или внешнему уровню вложенных операторов.

Ввод-вывод. В языке Ada не предусмотрены встроенные возможности ввода-вывода. Эти действия управляются перегруженными функциями из стандартных пакетов. Например, пакет Text_IO содержит определения простых функций ввода-вывода текстовых значений, необходимых для написания простых программ на языке Ada.

Функция get(arg) считывает числовой или строковый аргумент со стандартного устройства ввода (клавиатуры). Функция put(arg) записывает в стандартный вывод. Функция put(arg, fw) записывает значение параметра arg в поле длиной fw символов. Функция new_line записывает символ конца строки.

Преобразование между строковыми и числовыми данными также можно осуществить с помощью функций get и put. Функция get(in string, out number, out last) присваивает переменной number числовое значение первых 1..last символов строки string. Аналогично, функция put(out string, in number) присваивает строке string значение переменной number.

Функции get и put можно также использовать для файлов данных. Файлы определяются как limited private в компоненте private пакета, объявленного как определение реализации. Функцию open(in out File_Type, in File_Mode, in File_Name) можно использовать для открытия файла File_Name в одном из *режимов* (In_File, Out_File, Append_File), определяемом вторым параметром, а функцию close(in out File_Type) — для закрытия файла.

Указания компилятору. Оператор pragma используется для передачи информации компилятору. Этот оператор не рассматривается как часть языка. То есть если из правильной программы удалить все такие указания, программа должна по-прежнему компилироваться и иметь ту же семантику, что и раньше. Однако эти указания сообщают транслятору значимую информацию о том, как выполнять компиляцию для конкретного приложения.

Обычно подобное указание записывается следующим образом:

```
pragma имя_прагмы (параметры)
```

где имя_прагмы либо предопределено в языке, либо определено реализацией. Некоторые указания управляют функциями компилятора, такими как генерирование листинга программы (pragma LIST). Однако большинство указаний предназначено для управления структурами времени выполнения конкретных подпрограмм. Например, pragma INLINE(список_подпрограмм) определяет, что вызовы перечисленных подпрограмм везде, где возможно, должны быть заменены встроенными последовательностями кодов. Прагма pragma OPTIMIZE(TIME or SPACE) определяет, что под-

программа, в которой присутствует это указание, должна быть скомпилирована таким образом, чтобы минимизировать время ее выполнения (TIME) или необходимую для ее выполнения память (SPACE).

Исключительные ситуации. В конце каждой программной единицы (подпрограммы, блока, задачи или пакета) можно определить набор обработчиков исключительных ситуаций. Эти обработчики могут использоваться для обработки исключений, возникающих как внутри этой программной единицы, так и переданных из других подпрограмм, не имеющих своего обработчика ошибок. Каждое исключение имеет свое имя. Существует небольшое число предопределенных исключений, таких как CONSTRAINT_ERROR (возникает, когда индекс выходит за объявленный диапазон, когда при присваивании нарушается ограничение на диапазон значений и т. д.). Все остальные исключения объявляются следующим образом:

```
имя_исключения: exception
```

Обработчик исключений начинается с имен исключительных ситуаций, которые он обрабатывает, далее следует последовательность операторов, которые выполняют действия, необходимые для обработки исключения. Общий синтаксис обработчиков исключений следующий:

```
exception
        when имя_исключения | ... | имя_исключения
            => последовательность_операторов:
        ...
        when others => последовательность_операторов:
```

где каждая последовательность операторов может обрабатывать одну или несколько именованных исключений. Обработчик others других, не перечисленных ранее исключений, не обязателен. Однако если он присутствует, он обрабатывает все исключения, не перечисленные в предыдущих обработчиках.

Исключение генерируется либо неявно элементарной операцией, либо явно посредством выполнения оператора:

```
raise имя_исключения
```

Когда сгенерировано исключение, управление передается обработчику исключений текущей выполняемой программной единицы, если, конечно, эта программная единица имеет соответствующий этому исключению обработчик. Если обработчика нет, тогда исключение передается по цепочке вызовов подпрограмм первой из них, в которой имеется соответствующий обработчик, или, в конце концов, если нет программно-определенного обработчика, исключение передается на обработку системно-определенному обработчику. Исключения не передаются из задач.

Как только исключение обработано обработчиком, подпрограмма (или другая программная единица) нормально завершает свое выполнение и возвращает управление обратно в вызвавшую ее подпрограмму. Таким образом, обработчик исключений в языке Ada *завершает* выполнение тела подпрограммы, которое было прервано, когда возникло исключение. Здесь не предусмотрено возобновление выполнения программной единицы, в которой возникло исключение. Обработчик может частично обработать исключение, а затем передать это же исключение назад по динамической цепочке, выполнив оператор raise без имени исключения.

Задачи. Задача — это подпрограмма, которая может выполняться параллельно с другими задачами. Более подробно задачи были рассмотрены в разделе 11.2.

Встроенные приложения. Поскольку язык Ada разрабатывался для запуска встроенных приложений, в нем часто требуется специальный доступ к аппаратной части, на которой выполняются эти приложения. Ниже описаны некоторые из таких возможностей.

Задание адреса. С помощью следующего оператора при выполнении программы может быть определен конкретный адрес памяти, по которому следует поместить объект данных, подпрограмму или задачу:

```
for имя_объекта use at адрес_памяти
```

где адрес памяти задается как значение некоторого выражения.

Листинг П.2. Структура задач в языке Ada

```
task имя_задачи is
        - обьявления точек входа
end;
task body имя_задачи is
        - последовательность объявлений
    begin
        - последовательность операторов
    exception
        - обработчики исключений
end;
```

Прерывания. В случае встроенных систем часто бывает важно, чтобы программа могла обрабатывать прерывания, генерируемые аппаратной частью или внешними устройствами. С помощью следующей структуры можно связать прерывание с точкой входа задачи:

```
for точка_входа use at адрес_прерывания
```

Когда происходит прерывание, оно действует как вызов точки входа с наивысшим приоритетом для задачи с заданной точкой входа. Таким образом, задача, ответственная за обработку прерывания, может ответить немедленно, если она находится в состоянии ожидания при выполнении соответствующего оператора accept для этой точки входа.

Стандартные функции

В язык Ada включено множество стандартных пакетов, которые определяют обычные функции. Ниже перечислены некоторые из них:

Ada.Strings.Fixed	Функции строк фиксированной длины
Ada.Strings.Bounded	Функции строк переменной длины
Ada.Strings.Unbounded	Функции строк неограниченной длины
Ada.Numerics.Generic_Elementary_Functions	Тригонометрические функции

Пакеты

Пакеты языка Ada обеспечивают возможность инкапсуляции. Однако в язык Ada 95 добавлены дополнительные возможности для объектно-ориентированного наследования.

Определение пакета состоит из двух частей: *спецификации* (состоящей из *видимой части* и *закрытой части*) и *тела*. Спецификация содержит информацию, необходимую для правильного использования пакета, тело содержит как инкапсу-

лированные локальные переменные и определения подпрограмм, так и тела подпрограмм, вызываемых извне пакета. В некоторых случаях пакет не нуждается в теле. Стандартная форма определения пакета приведена в листинге П.3.

Листинг П.3. Спецификация пакета Ada

```
package имя_пакета is
        - обьявления видимых обьектов данных
private
        - полное определение закрытых обьектов данных
end;
package body имя_пакета is
        - определения объектов данных и подпрограмм,
        - обьявленных выше в части спецификации пакета
begin
        - операторы инициализации пакета
        - при первом создании его экземпляра
exception
        - обработчики исключений
end;
```

Видимая часть спецификации пакета (все, что предшествует ключевому слову `private`) определяет, какие объекты данных внешний пользователь пакета может использовать в программе. Если определения типов даны в видимой части, пользователь может объявлять переменные этого типа; если там определены переменные или константы, на них можно ссылаться и т. д.

Если в пакете определяется абстрактный тип данных, пользователь может объявлять переменные, принадлежащие к этому абстрактному типу, но подробное описание типа не должно быть видимым. В этом случае определение типа в видимой части имеет вид

```
type имя_типа is private;
```

а полное определение задается в закрытой части спецификации (например, строки 6–12 в листинге П.1).

Казалось бы, логично поместить закрытые компоненты пакета в его тело, а не в спецификацию, чтобы спрятать такие имена от других пакетов. Однако по соображениям, связанным с реализацией, это делается не так. Более подробно этот момент обсуждается в разделе 7.1.

Ограниченные закрытые типы. Обычно желательно, чтобы элементарные операции присваивания и проверки на равенство и неравенство были автоматически доступны даже для объектов закрытых типов, так что операция `A := B` была бы определена для объектов `A` и `B` закрытого типа. В языке Ada это единственные операции, которые предусмотрены для закрытых типов без явного определения программистом. Однако во многих случаях даже эти операции не должны быть заранее определены для закрытого типа. Если программист описывает тип как `limited private` (ограниченный закрытый тип), то никакие операции вообще не будут заранее определены для этого типа.

Общие пакеты. Спецификация общего пакета имеет параметры, задающие типы конкретных элементов пакета, размер конкретных массивов пакета и т. д. Например, общий пакет, который определяет абстрактный тип `stack`, может иметь параметр, задающий тип компонента, сохраняемого в стеке. В общем случае это выглядит следующим образом:

```
package имя_фактического_пакета is
        new имя_общего_пакета(параметры);
```

Создание экземпляра пакета — это операция, выполняемая компилятором. Во время компиляции в операторе, который создает экземпляр пакета, каждая ссылка на имя формального параметра в пакете заменяется фактическим параметром. Получившийся пакет может быть скомпилирован так же, как и обычный пакет. Общий пакет и созданный его экземпляр не влияют на структуру времени выполнения, только результирующий пакет (после создания экземпляра) действительно компилируется в объекты данных времени выполнения и в выполняемый код. Однако, поскольку создание экземпляров общих пакетов в языке Ada — достаточно часто встречающаяся операция, особенно в тех программах, в которых используются библиотечные пакеты, большинство компиляторов Ada различными способами оптимизируют эту операцию. Часто это осуществляется путем частичной предварительной компиляции общего пакета, когда не компилированными остаются лишь те части, которые напрямую зависят от параметров общего пакета и которые только и будут компилироваться при каждом создании экземпляра пакета.

Наследование

Язык Ada 95 расширил концепцию пакета, включив возможность наследования. Ранее нужно было с помощью конструкции with включать явные пакеты или использовать производные типы. В язык Ada 95 добавлено понятие *расширение типа*, которое осуществляется с помощью нового, названного помеченным (tagged), типа. Например, если бы тип Mydata из листинга П.1 был бы описан как

```
type Mydata is tagged record
        val: myarray;
        sz: integer := 0;
        end record;
```

то можно было бы определить новый объект Doublestack:

```
type Doublestack is new Mydata with
        record
        Newdata: myarray;
        end record;
```

Этот объект Doublestack имел бы унаследованные от типа Mydata компоненты val и sz. Использование помеченных типов в языке Ada 95 позволяет создавать новые пакеты без необходимости повторной компиляции определений существующих пакетов.

Каждый помеченный тип содержит атрибут class (например, class в Mydata'class или Doublestack'class, который может быть использован для динамического связывания). Объект Doublestack является членом того же класса, что и Mydata, поскольку произведен из него.

Помеченные объекты также можно объявлять как абстрактные (abstract). Например,

```
type Mydata is abstract tagged with null record
```

задает имя объекта, обозначение его класса, но не определяет способ представления этого объекта в памяти. Определение объекта Mydata может быть заполнено с помощью расширения типа. Таким образом, создается шаблон для типа, а для использования объекта должны быть определены помеченные типы.

Вызов элементарной операции с фактическим параметром помеченного типа приводит к связыванию во время выполнения с фактической подпрограммой, которая должна быть вызвана. Это обеспечивает наличие в Ada 95 существенных свойств динамической объектно-ориентированной модели выполнения, что и отличает Ada 95 от Ada 83.

П.2. С

Пример с пояснениями

В листинге П.4 приведен пример программы на языке С, производящей суммирование элементов массива. В данном примере для входных данных

```
41234
0
```

программа напечатает:

```
1 2 3 4 : SUM= 10
```

Листинг П.4. Пример программы суммирования элементов массива на языке С

```
 1 #include <stdio.h>
 2 const int maxsize=9;
 3 main()
 4     {int a[maxsize];
 5     int j,k;
 6     while( (k=convert(getchar())) != 0) {
 7         for (j=0; j<k; j++) a[j] = convert(getchar());
 8         for (j=0; j<k; j++) printf("%d ", a[j]);
 9         printf (": SUM= %d\n", addition(a,k));
10         while(getchar() != '\n');
11         } }
12 /* Функция преобразования */
13 int convert (char ch)
14     {return ch-'0';}
15 /* Функция сложения */
16 int addition (v, n)
17     int v[ ], n;
18     { int sum,j;
19     sum=0;
20     for (j=0; j<n; j++) sum=sum+v[j];
21     return sum;
22 }
```

Строка 1. В данной программе должна быть использована стандартная библиотека ввода-вывода, подключаемая через заголовочный файл stdio.h. В этой строке записана препроцессорная команда, которая обрабатывается до начала трансляции. В этом месте происходит включение в программу содержимого файла stdio.h как части программы.

Строка 2. Определяется целая константа maxsize, и ей присваивается значение 9. Можно было использовать также препроцессорную команду с похожим результатом:

```
#define maxsize 9
```

Строка 3. В этом месте начинается вычисление. Некоторая функция должна иметь имя main.

Строка 4. Объявляется массив a целых чисел с индексом из диапазона значений от 0 до 8.

Строка 6. Функция getchar() определена в файле stdio.h. Она считывает очередной символ со стандартного ввода. Поскольку переменные символьного типа char являются подтипами целых чисел, читается целочисленное значение (код) символа. Например, в системах, использующих ASCII-коды, цифра 2 является символом с двоичным значением 0110010, которое в восьмеричной системе равняется 62, а в десятичной — 50. Однако в данной программе требуется численное значение 2, которое и получается в результате действия функции convert() (строки 13–14).

Считанное значение присваивается переменной k. Обратите внимание на то, что присваивание здесь является просто операцией, а выражение k=convert(getchar()) присваивает следующее введенное число переменной k и возвращает его же в качестве значения этого выражения. Если это значение не 0, то выполняются следующие операторы (строки 7–11). Следовательно, если значение k равно 0, то цикл завершится, а вместе с ним завершится и программа, так как закончится выполнение процедуры main.

Строка 7. В операторе for последовательно считываются значения массива (с помощью функции convert и функции getchar из стандартной библиотеки ввода-вывода). Первым параметром оператора for является операция присваивания значения переменной (счетчику цикла), выполняемая при входе в этот оператор цикла; обычно присвоенное значение является начальным значением для цикла (j=0). Второй параметр является условием продолжения выполнения цикла (выполнять цикл, пока j<k), а третий параметр описывает операцию, выполняющуюся в конце каждого шага цикла (j++, то есть добавление единицы к значению переменной j).

Строка 8. Этот оператор for аналогичен предыдущему оператору цикла, но только на каждом шаге цикла печатает соответствующий элемент массива. Функция printf выводит на печать свой первый аргумент, представленный символьной строкой. Последовательность символов %d в строке вывода указывает, что нужно взять следующий аргумент функции printf и напечатать его в целочисленном формате в том месте строки вывода, где находится эта последовательность.

Строка 10. Программа читает символы до тех пор, пока не встретится символ конца строки. Данный оператор while имеет пустое тело.

Строка 11. Фигурные скобки завершают блок, начинающийся в строке 6, и тело основной программы, которое начинается в строке 4.

Строка 12. Комментарий. Его можно использовать везде, где можно поместить символ пробела.

Строка 14. Не обязательно знать ASCII-коды для цифр. Все, что необходимо знать, это то, что символ '0' — это число 0, символ '1' — на единицу больше символа '0' и имеет значение 1, а '2' — больше '0' на 2 и имеет значение 2 и т. д. Выражение ch-'0' вернет требуемое числовое значение для одной цифры.

Строки 16–17. Здесь приведен старый стиль описания параметров языка C. Имена параметров находятся в списке параметров, а затем в теле функции следует описание их типов. Обратите внимание на то, что границы массивов не заданы.

П.2.1. Объекты данных

Язык С является строго типизированным языком, но в нем существует всего несколько типов, поэтому строгая типизация полезна лишь в минимальной степени. Данные могут быть целочисленными (integer), перечисляемого типа или вещественными с плавающей точкой (float). Структурированные данные представлены массивами, строками (форма массивов) или записями, такими как struct или union.

Элементарные типы данных

Переменные и константы. Идентификаторами в С могут быть любые последовательности букв, цифр и символа подчеркивания (_), причем они не должны начинаться с цифры. В заголовочных файлах .h обычно используются внутренние переменные, имена которых начинаются с символа подчеркивания, так что в пользовательских программах лучше избегать задания идентификаторов, начинающихся с символа подчеркивания. Регистр играет важную роль в языке С, поэтому переменные с именами abc и ABC — это разные переменные.

Целые числа можно задавать в обычном формате целых чисел (например, 12, -17) или как восьмеричные литералы с предшествующим 0 (например, 03 — десятичное 3, 011 — десятичное 9, 0100 — десятичное 64). Символьные литеральные константы задаются в одинарных кавычках (например, 'A', 'B', '1'). Символ \n обозначает конец строки, а \0 — это *null*-символ, который используется при обработке строк. Действительные ASCII-коды символов можно задавать, используя числовые коды. Так, \062 соответствует ASCII-коду 62 (восьмеричному) или символу '2'.

Дискретные значения можно создавать с помощью *перечисляемых типов* — другой формы целочисленных данных:

```
enum имя_типа { список_значений } имя
```

Пример:

```
enum colors { red, yellow, blue } x, y, z;
x = red;
y = blue;
```

Значения перечисляемого типа нужно указывать только один раз. Следующее объявление переменных перечисляемого типа colors можно записать как

```
enum colors a, b, c;
```

Литеральные константы с плавающей точкой могут быть записаны в форме числа с десятичной точкой (например, 1.0, 3.456) или в экспоненциальной форме (например, 1.2e3 = 1200, 12e-1 = .012).

Строковые литералы, или просто строки, задаются как последовательности символов, заключенные в двойные кавычки ("), например, "abc", "1234". Однако строки на самом деле являются строковыми массивами со специальными завершающими null-символами (\0). Таким образом, строка "123456" на самом деле является последовательностью 123456\0 и обрабатывается так же, как если бы она была объявлена одним из следующих способов:

```
char arrayvar[] = "123456"; /* arrayvar длиной 7 символов */
char *arrayptr = "123456";  /* arrayptr указывает на эти символы */
```

Числовые типы данных. Целый тип в языке С имеет несколько разновидностей. Целый тип short содержит наименьшее количество бит, эффективное для

данной реализации. Для многих реализаций С, используемых на персональных компьютерах, этот тип соответствует 16-битному целому числу со знаком, максимальное значение которого может быть 32 767. Он может быть и больше, например 32-битное целое. Тип long является, наоборот, наиболее длинной (в отношении количества бит) реализацией целых чисел, приемлемой для данной конкретной аппаратной части компьютера. Как правило, переменные этого типа занимают 32 бита, а иногда и больше. Тип int представляет собой наиболее эффективную реализацию для данной аппаратной части. Этот тип (int) может совпадать с short или long или его длина может иметь некоторое промежуточное значение в интервале между значениями длин описанных типов.

Как показано в приведенном примере, тип char также является подтипом целого, в данном случае — однобайтным значением. То есть любыми введенными символами можно манипулировать так же, как другими целыми значениями.

В языке С нет булевого или логического типа данных; вместо них также используются целые переменные. Истина определена как любое ненулевое целое значение, а Ложь представляется нулевым значением. Для установки и сброса различных битов в целых числах можно использовать поразрядные операции. Хотя для хранения различных булевых значений можно использовать несколько битов в отдельном слове (например, двоичное число 101 = десятичное 5 может обозначать «А — истинно, В — ложно, С — истинно»), для этой цели обычно лучше использовать отдельные переменные типа int.

Константы можно определять либо при помощи препроцессорной команды #define, как было показано выше, либо при помощи объявления const:

```
const имя_типа = значение
```

Указатели. Указатели могут указывать на любые данные. Конструкция *p показывает, что p является указателем на объект. Например, операторы

```
int *p:
p = (int *) malloc(sizeof(int)):
p = 1:
```

делают следующее:

1. Размещают в памяти указатель p, который будет указывать на целую переменную.
2. Отводят достаточно памяти, чтобы разместить там целую переменную, и сохраняют *l*-значение этой области памяти как *r*-значение переменной p.
3. Используя *r*-значение переменной p, помещают целое значение 1 в отведенное пространство памяти, на которое указывает p.

Структурированные типы данных

Массивы. Индексы одномерных массивов начинаются с 0 (например, int a[3]). Многомерные массивы любой размерности можно определить как

```
int a[10][5]
```

Такие массивы сохраняются путем *развертывания по строкам*, то есть как если бы массив был описан следующим образом:

```
thing a[10]
```

где тип thing — это целый массив длины 5.

Как говорилось ранее, строки являются массивами символов. Строковые функции обычно используют строки с завершающим null-символом (то есть в виде массивов символов, завершающихся null-символом: "abcdef" = abcdef\0).

Определяемые пользователем типы

Язык С может создавать записи структурированных данных, называемые *структурами*, с помощью типа struct. Синтаксис структур похож на синтаксис перечисляемых констант, описанных выше. Например,

```
struct MarylandClass {int size: char instructor[20]:} CMSC330. CMSC630:
```

определяет структуру MarylandClass, содержащую целый компонент size и компонент instructor в виде массива из 20 символов, и объявляет две переменные CMSC330 и CMSC630 типа MarylandClass. А следующий оператор:

```
struct MarylandClass CMSC430:
```

определяет новую переменную CMSC430 типа MarylandClass, ссылаясь на то же самое описание структуры. К членам структуры доступ осуществляется с использованием точечной нотации. Так, CMSC330.size будет целочисленным компонентом size этой структуры, а соответствующее имя CMSC330.instructor будет компонентом instructor этой же структуры.

Объявление записи с помощью конструкции struct создает новый тип. Парадоксально, но объявление typedef на самом деле является просто подстановкой имени, но новый тип оно не создает. Соответственно

```
typedef int newint:
```

определяет, что тип newint является таким же, как и тип int. Обычно такое описание используется при определении структур struct. Так, во фрагменте

```
typedef struct NewClass { ... } MarylandClass:
MarylandClass A:
MarylandClass B:
```

определяется новое имя MarylandClass для структурного типа NewClass, и с помощью нового имени типа переменные A и B объявляются как принадлежащие также к структурному типу NewClass.

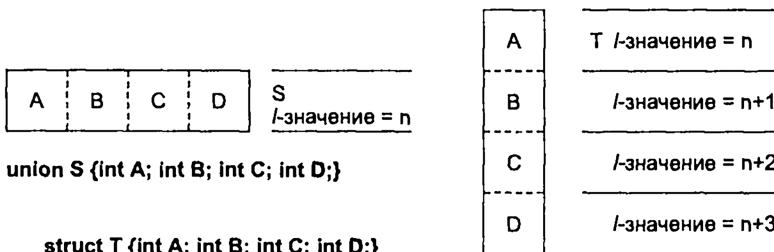


Рис. П.1. Память, выделяемая для объединения S и структуры T

Объединение. Объединение (union) — это определение типа, синтаксически похожее на определение типа struct, но семантически эквивалентное вариантным записям языка Pascal (раздел 6.1.6). Различие между типами union и srtuct показано на рис. П.1. В объединении S каждый его компонент имеет *l*-значение такое же,

как и *l*-значение самого объединения S. Каждый компонент располагается в одной и той же области памяти. В структуре T *l*-значение компонента A то же самое, что и у нее самой, но все *l*-значения остальных компонентов отличны от *l*-значения структуры T. Каждый компонент располагается в своей собственной области памяти.

Представление объектов в памяти. Целые, вещественные с плавающей точкой и символьные данные хранятся в своем естественном формате, а массивы не требуют дескрипторов. Указатели являются просто *l*-значениями объектов, на которые они указывают. Накладные расходы при доступе к этим объектам незначительны.

Указатели и массивы тесно связаны. К массиву

```
int A[10]
```

также можно получить доступ как к указателю:

```
int *arrayptr;
arrayptr = &A[0]; /* arrayptr указывает на A[0] */
```

Выражение *((arrayptr) + (i)) имеет то же значение, что и a[i].

Если q — это структура struct:

```
struct q { int a; int b; }
```

тогда последовательность

```
struct q *p;
p = (struct q *) malloc(sizeof(struct q));
```

как и в предыдущем примере, выделяет память для структуры q и сохраняет ее адрес в переменной-указателе p. Для доступа к компонентам этой структуры используется операция → (которая записывается как ->). На компонент a ссылаются следующим образом: p → a; а на компонент b — p → b. Обратите внимание на то, что p → a есть *в точности* то же самое, что и (*p).a.

Инициализация. Любую статически размещаемую переменную можно инициализировать следующим образом:

```
int i=12;
```

Если нужно инициализировать массив, то задается список элементов:

```
int a[4] = (1, 2, 3, 4); char string[4] = "abc";
```

В последнем примере массив string инициализируется как abc\0. В массиве должно быть зарезервировано место под завершающий null-символ.

П.2.2. Управление последовательностью действий

Выражения

Одна из сильных сторон языка C — это наличие множества операций, которые могут обрабатывать числовые данные. (Одновременно это является и его слабым местом, потому что теперь существует много способов выполнить аналогичные операции.) Множество операций языка C и уровни их приоритетов были приведены ранее, в табл. 8.2.

Важно помнить о различии между поразрядными и логическими операциями. Операция поразрядного логического И в выражении 5&4 приведет к результату, рав-

ному значению 4, поскольку эта операция выполняет логическое И для каждого бита отдельно:

```
5 & 4 = 0101 & 0100 = 0100 = 4
```

Операция логического И в выражении 5 && 4 приведет к значению 1, поскольку она выполняется следующим образом:

```
a && b =  if a = 0 then 0 else if b = 0 then 0 else 1
```

Другие логические операции выполняются подобным же образом. Они всегда возвращают значение 0 или 1.

Выражение if (a = b) присваивает значение b переменной a и возвращает значение a. Будьте внимательны — если вы имеете в виду проверку равенства двух переменных, используйте if (a==b).

Приведение типов. Обычно приведение типов встречается тогда, когда не происходит потери информации, как при переходе от типа char к типу int. Приведение значения к новому типу можно осуществить, если поставить перед ним унарную *операцию приведения типа*. Так, a + (int)b сначала приведет b к типу int, а потом прибавит b к a.

Операторы

Блоки. Последовательность {список_операторов} можно использовать везде, где необходим оператор. Также в каждом блоке можно объявить локальные переменные:

```
{int i, j; ...}
```

Однако, как отмечалось в разделе 9.4.2, память под переменные i и j будет отведена в тот момент, когда будет отводиться память под всю процедуру, содержащую этот блок.

Выражения-операторы. Любое выражение можно использовать в качестве оператора. В частности, оператор присваивания a = b в действительности есть выражение присваивания.

Условные операторы. В языке С имеются обычные конструкции if then и if then else:

```
if then:      if (выражение) оператор
if then else: if (выражение) оператор else оператор
```

Во вложенных операторах if оператор else ассоциируется с ближайшим к нему if.

Операторы цикла. В языке С существует три оператора цикла — while, do и for:

- while: while(выражение) оператор: означает, что следует выполнять оператор до тех пор, пока выражение остается истинным.
- do: do оператор while(выражение); означает выполнить оператор, а затем проверить выражение. Если оно истинно, то повторить оператор do. Таким же оператором цикла является оператор repeat until в языке Pascal и некоторых других языках; отличие заключается лишь в том, что цикл продолжается, если тестовое выражение истинно.
- for: for(выражение$_1$; выражение$_2$; выражение$_3$) оператор: является формой итерации и выполняется следующим образом.

  1. Если задано выражение$_1$, то оно вычисляется. Часто (но не обязательно) это оператор инициализации, например J=0.

2. Если задано $выражение_2$, то оно вычисляется. Если результат равняется 0, то цикл for завершается. Обычно $выражение_2$ определяет условие прекращения цикла, например J<10.
3. Выполняется оператор.
4. Если задано $выражение_3$, то оно вычисляется. Обычно это приращение переменной — счетчика цикла, например J++ или просто прибавление 1 к J.
5. Процесс повторяется с шага 2.

Оператор switch. Действие оператора switch подобно многовариантному ветвлению; часто он называется оператором case. Его синтаксис следующий:

```
switch(выражение)
      {case константа1: список_операторов1: break:
      case константа2: список_операторов2: break:
      case константа3: ...:
      default: список_операторовn:}
```

После того как вычисляется выражение, управление переходит к метке case со значением константы, равной вычисленному значению выражения, или к оператору default, если ни одна константа не совпадает с полученным значением выражения.

В данном случае очень важен оператор break, и это является действительно слабым местом языка С, поскольку если отсутствует явная передача управления после операторов одной метки case, то будут выполнены операторы и всех последующих меток case.

Операторы передачи управления. В языке С существуют четыре оператора, передающих управление, — это break, continue, goto и return.

Оператор break вызывает завершение ближайшего к нему из операторов while, do, for или switch, в которые он вложен. По существу, это оператор goto, передающий управление первому оператору, который следует за составным оператором, содержащим оператор break. Как отмечалось ранее, этот оператор очень важен для предотвращения последовательного продолжения выполнения ветвей case в операторе switch.

Оператор continue передает управление в конец тела цикла непосредственно содержащего его оператора for, do или while. Это приводит к тому, что программа переходит к следующей итерации соответствующего цикла.

Оператор goto метка передает управление оператору, помеченному указанной меткой. Как и в языке FORTRAN, это слабый оператор, его использование необязательно и лучше им вообще не пользоваться. Вложенные в операторы цикла операторы break и continue обеспечивают все необходимые механизмы передачи управления.

Оператор return осуществляет возврат из процедур. Если процедура вызывалась как функция, то синтаксис выглядит следующим образом: return выражение.

Команды препроцессора. Ключевые слова define, ifdef, ifndef, include, if, undef и else, начинающиеся с символа #, являются командами или директивами препроцессора и не имеют никаких других функций в языке С. В старых версиях транслятора С символ # обязательно должен был являться первым символом в строке, однако в большинстве современных трансляторов перед этим символом допускаются пробелы.

Директива #define именует последовательность лексем, например:

```
#define имя последовательность_лексем
```

Именованные константы могут быть определены следующим образом:

```
#define TRUE 1
```

Обратите внимание на то, что запись

```
#define TRUE = 1
```

неправильна, поскольку в этом случае значение константы TRUE будет = 1, а не 1.

Директиву #define также можно использовать для создания макроопределений, например:

```
#define имя(Var1, Var2, ..., Varn) последовательность_лексем
```

где Var_i при использовании заменяются истинными аргументами, например:

```
#define abs(A, B) A<B ? A : B
```

abs теперь можно использовать в любом выражении. Например, выражение abs(newvar, oldvar) после обработки макроопределения преобразуется в выражение newvar < oldvar ? newvar : oldvar, которое и будет транслироваться компилятором.

Директива #include добавляет в программу текст из соответствующего файла:

```
#include < имя_файла >
#include "имя_файла"
```

Первая директива добавляет файл с указанным именем из системной библиотеки, которая в системе UNIX обычно находится в каталоге /usr/include. Вторая директива добавляет файл из того же каталога, в котором расположен файл, содержащий текст исходной программы. Существует соглашение, по которому все определения интерфейсов хранятся в заголовочных файлах .h и при необходимости копируются директивой #include.

Директива #ifdef используется для проверки, было ли ранее определено данное имя. Если было, то в исходный текст программы добавляется последовательность операторов

```
#ifdef имя_переменной
операторы
#endif
```

Аналогично директива #ifndef добавляет операторы, если имя не было ранее определено.

Директива #if добавляет текст в зависимости от значения заданного в ней константного выражения.

Директива #undef делает имя более не определенным: #undef имя_переменной.

Директива #else используется в качестве предложения else в директивах #if, #ifdef, #ifndef в случае, если не выполнено условие, заданное в них самих.

Ввод и вывод

В языке С не существует специальных операторов ввода и вывода. Ввод и вывод осуществляется с помощью заранее указанного набора функций, которые определены в заголовочном файле stdio.h. Ввод и вывод описаны в следующем разделе «Стандартные функции».

Стандартные функции

Сила языка С заключена в богатой библиотеке функций, которые оказывают неоценимую помощь программисту при написании программы. Многие из них напрямую связаны с операционной системой. В данной книге нет возможности описать их все. Ниже приведены лишь те функции, которые наиболее важны для написания простых программ.

Стандартный ввод и вывод. Определения этих функций находятся в файле stdio.h. Для простых программ ввода и вывода наиболее важными являются getchar (для ввода символов) и printf (для простого форматированного вывода).

Файлы stdin и stdout заранее определены соответственно в качестве стандартного входного файла (обычно это клавиатура) и стандартного выходного файла (обычно это экран монитора). Эти имена в списке параметров могут быть просто опущены. Например, для считывания символов из файла требуется функция getc(filename), в то время как для считывания с клавиатуры необходимо просто getchar без имени файла. Функция getchar определена просто как getc(stdin).

Определена именованная константа EOF, которая используется для обозначения достижения конца файла. (Обычно это значение устанавливается равным -1, чтобы отличить его от всех стальных считываемых символов.)

Функция int getchar() возвращает очередной символ из входного потока (stdin).

Функция int getc(FILE *имя_файла) возвращает следующий символ из файла с указанным именем.

Функция FILE * fopen(char *filestring. char *filetype) открывает файл filestring для ввода (если filetype = "r"), для нового вывода (если filetype = "w") или для добавления к существующему файлу (если filetype = "a"). Файлы stdin (стандартный ввод) и stdout (стандартный вывод), а также stderr (вывод ошибок) открываются автоматически.

Функция putchar(char x) печатает символ x в файл stdout.

Функция putc(char x. FILE *имя_файла) выводит символ x в файл с указанным именем.

Функция fgets(char *s. int n. FILE *имя_файла) считывает массив символов из заданного файла, s является указателем на этот массив. Функция считывает символы до тех пор, пока:

1) не встретит новую строку;
2) не достигнет конца файла;
3) не считает n-1 символ.

Завершающий null-символ \0 добавляется к строке, на которую указывает s.

Функция feof(FILE *имя_файла) возвращает истину, если при предыдущем чтении данного файла был достигнут его конец.

Функция int printf(строка. аргументы) печатает указанную строку в стандартный вывод stdout. Если после вывода строки на терминале следующий вывод должен начаться на новой строке терминала, то строка должна заканчиваться символом конца строки \n. Если в строке содержится %d или %i, то следующий аргумент из списка параметров печатается в целочисленном формате. Если содержится %s, то предполагается, что следующий аргумент — это строка, завер-

шающаяся null-символом \0, которую требуется распечатать. %c означает вывод первого символа следующего аргумента, %f — формат вывода вещественного числа без явной экспоненты, а %e — формат вывода на печать вещественного числа в экспоненциальной форме. %o означает вывод данных в восьмеричном формате.

Функция int sprintf(char *s. строка. аргументы) действует аналогично printf, но строка пишется в то место памяти, на которое указывает s.

Функции распределения памяти. Эти функции определены в заголовочном файле malloc.h.

Функция void *malloc(целочисленное_значение) отводит блок памяти, размер которого равен указанному целочисленному значению, и возвращает указатель на него.

Функция int sizeof(имя_типа) возвращает размер объекта указанного типа. Часто используется вместе с malloc, например:

```
ptrvar = (newtype*) malloc(sizeof(newtype));
```

Функция int free(char mallocptr) восстанавливает область памяти (на которую указывает mallocptr), ранее выделенную под некоторый объект с помощью функции malloc.

Строковые функции. Функция char *strcat(char *s1. char *s2) добавляет строку s2 в конец строки s1 и возвращает указатель на s1.

Функция char *strncat(char *s1. char *s2. int n) добавляет не более n символов из строки s2 (или всю строку s2, если она короче, чем n) в конец строки s1 и возвращает указатель на s1.

Функция int strcmp(char *s1. char *s2) возвращает значение, меньшее 0, если строка s1 лексикографически меньше, чем s2; значение 0, если они равны; и значение большее 0, если s1 больше, чем s2.

Функция int strncmp(s1. s2. n) похожа на strcmp, за исключением того, что сравнение производится только до *n*-го символа s2.

Функция char *strcpy(char *s1. char *s2) копирует строку s2 в строку s1. Указатель возвращается на s1.

Функция char *strncpy(char *s1. char *s2. int n) копирует первые n символов s2 в s1.

Функция int strlen(char *строка_символов) возвращает длину строки.

Функции преобразования. Функция double strtod(char *строка_символов. char **ptr) преобразует содержимое строки к типу float и устанавливает указатель ptr на символ, следующий за последним преобразованным символом. Если указатель ptr имеет значение null, это значит, что из строки не удалось выделить вещественное число.

Функция long strtol(char * строка_символов. char **ptr. int base) преобразует содержимое строки в длинные целые числа (long). Здесь base — это основание системы счисления вводимых данных. Чаще всего используются значения 2, 8, 10.

Функции atoi(char *строка_символов) (преобразование кодов ASCII в значения типа int), atol (ASCII — в long) и atof (ASCII — в float) остались от старых версий языка С.

П.3. C++

Пример с пояснениями

В листинге П.5 представлена С++-версия программы из раздела П.2, написанной на языке С, которая считывает массив целых чисел и выводит на печать сумму элементов этого массива. Возможно, этот пример чуть более сложный, чем необходимо для того, чтобы продемонстрировать использование классов, наследования и потоков ввода-вывода.

Листинг П.5. Пример суммирования элементов массива на языке С++

```
 1 # include <stream.h>
 2    // Это потоки ввода-вывода C++. stdio.h также работает.
 3 class DataConvert {
 4 protected:
 5 int convert(char ch) {return ch-'0';}};

 6 class DataStore: DataConvert{
 7 public:
 8 int initial(char a)
 9     {ci=0:
10     return size = convert(a):}:
11 void save(char a)
12     {store[ci++]=convert(a):}:
13 int setprint() { ci=0: return size:}:
14 int printval() { return store[ci++]:}:
15 int sum()
16     {int arrsum:
17     arrsum=0:
18     for(ci=0:ci<size:ci++)arrsum=arrsum+store[ci]:
19     return arrsum:}
20 private:
21 const int maxsize=9:
22 int size:  //Размер массива
23 int ci :   //Текущий индекс массива
24 int store[maxsize]:}:

25 main()
26     { int j.k:
27     DataStore x:
28     while((k=x.initial(cin.get()))!=0)
29         {for(j=0:j<k:j++)x.save(cin.get()):
30         for(j=x.setprint():j>0:j--)cout << x.printval():
31         cout << ": SUM=" << x.sum() << endl:
32         while(cin.get()!='\n'):}}
```

Строка 1. Заголовочный файл stream.h[1] содержит объявления функций стандартной библиотеки ввода-вывода, в том числе функций cin и cout. Также можно использовать стандартную библиотеку языка С, подключаемую через заголовочный файл stdio.h.

[1] При использовании компилятора С++ фирмы Microsoft заголовочный файл называется iostream.h. — *Примеч. науч. ред.*

Строка 2. В языке С++ комментарии выглядят подобно комментариям в Ada. Они начинаются с символов // и распространяются до конца строки. Также можно использовать комментарии языка С вида /* ... */.

Строки 3–5. Определяется класс DataConvert.

Строка 4. Все объекты, объявляемые в классе DataConvert, будут иметь класс доступа protected (то есть защищенный). Это означает, что функция convert в строке 5 будет видна в любом классе, производном от класса DataConvert.

Строка 5. Функция convert преобразует символ, представляющий цифру, в его числовой эквивалент (например, convert('1') = 1, convert('3') = 3, ...) так же, как и в программе на языке С, приведенной в листинге П.4.

Строка 6. Класс DataStore является производным от класса DataConvert. Функция convert, объявленная в классе DataConvert (записанная в виде DataConvert::convert), видна в этом определении класса.

Строка 7. Ключевое слово public (открытый) означает, что объявления в строках с 8-й по 19-ю видны вне определения класса.

Строки 11–12. Класс DataStore создает абстракцию данных для хранимых данных, инкапсулируя фактические значения данных в массив, который не виден вне определения класса.

Строка 13. Функция setprint возвращает число элементов, которые нужно напечатать, а затем многократно вызывается printval для последовательного вывода на печать значений.

Строка 14. Функция printval возвращает следующее значение из массива. Эту функцию необходимо вызывать, поскольку массив store инкапсулирован в классе DataStore и доступ к нему имеют только функции, определенные в данном классе. Если бы массив store был открытым, то не требовалось бы использовать эту функцию.

Строка 20. Все объявления, которые следуют за ключевым словом private, будут закрытыми и могут использоваться только внутри определения этого класса.

Строка 25. Начало основной программы main.

Строки 26–27. Переменные j и k являются локальными. Объект x класса DataStore является нашим объектом для хранения массива.

Строка 28. Эта строка аналогична строке 6 листинга П.4. Последовательность действий следующая:

1) для возврата одиночного символа вызывается функция get класса cin, определенного в заголовочном файле stream.h;
2) для преобразования размера массива и сохранения этого значения в переменной x.size вызывается функция DataStore::initial (строка 8);
3) размер массива устанавливается равным k, и если k не равно 0, то осуществляется обработка массива.

Строка 29. Оператор for вызывает функцию get для чтения каждого вводимого пользователем числа в массиве и сохраняет его в x.store посредством вызова функции DataStore::save.

Строка 30. Функция DataStore::setprint инициализирует оператор for, который последовательно уменьшает значение счетчика цикла до 0 и выводит на печать каждый элемент массива посредством вызова функции DataStore::printval.

Потоковая функция cout осуществляет конвейерную пересылку данных в функцию. Таким образом, чтобы напечатать a, b и c, необходимо написать либо:

```
cout << a << b << c
```

либо

```
cout << a
cout << b
cout << c
```

Строка 31. С помощью вызова функции DataStore::sum на печать выводится сумма массива. Объект endl, который по конвейеру передает функция cout, добавляет символ конца строки и выводит строку на печать.

Строка 32. Функция get класса cin вызывается в цикле до тех пор, пока не встретится символ конца строки \n. Это необходимо для того, чтобы пропустить весь ввод пользователя в строке, когда уже прочитаны все элементы обрабатываемого массива.

П.3.1. Объекты данных

Элементарные данные в языке C++ такие же, как и в языке C.

Элементарные типы данных

Так же как и в языке C, в C++ используются элементарные типы int и float.

Определяемые пользователем типы

Помимо int и float, для создания структурных объектов в программе можно использовать такие конструкции языка C, как struct и typedef.

Кроме того, в языке C++ существует возможность определять классы с помощью конструкции class, которая является расширением struct за счет добавления дополнительных компонентов структуры — подпрограмм (*методов*).

Синтаксис описания класса приведен ниже:

```
class имя_класса
      {компонент_класса₁:
      компонент_класса₂:
      ...
      компонент_классаₙ: }
```

Здесь каждый компонент класса является либо объявлением объекта данных, либо объявлением метода.

Объявления объектов данных могут быть либо объявлениями const, либо объявлениями структур struct, либо объявлениями объектов данных любого типа или класса. В языке C++ нет необходимости явно использовать ключевое слово struct при объявлении переменной данного типа. Так, для структуры struct Myclass объекты данных можно описывать следующим образом:

```
Myclass A, B, C;
```

Объявление методов — это определение функций или процедур, которые могут вызываться как операции данного класса. Можно задавать полное определение или просто заголовок сигнатуры функции, в котором определяются все ее параметры. Этот второй способ похож на определение пакета в языке Ada, где детали реализации функции могут быть определены в любом месте в теле пакета. Такая заголо-

вочная информация необходима для генерации кода вызова функции вне определения класса.

Каждый член класса имеет атрибут *доступа*, который описывает видимость этого члена. Для каждого компонента класса можно задать один из трех возможных атрибутов доступа. Атрибут public (открытый) обозначает, что данное имя известно за пределами класса. Атрибут protected (защищенный) подразумевает, что данное имя может использоваться только в производных классах. Третий атрибут private (закрытый) указывает, что данное имя может использоваться только внутри класса, содержащего его объявление.

Атрибут доступа предшествует наименованию компонента класса, как в следующем компоненте

```
protected: void setvalue(int *a, int b) { *a = b; }
```

который присваивает, используя передачу параметра по ссылке, *r*-значение параметра b объекту, переданному как *l*-значение указателя a.

Однажды заданный, атрибут доступа остается атрибутом по умолчанию, пока не будет изменен. Если атрибут доступа не указан, по умолчанию используется атрибут private.

Дружественные классы. Атрибуты доступа protected и private иногда создают больше ограничений, чем необходимо. Иногда некоторый класс может позволить другому классу получить доступ к своим закрытым данным. Эта возможность осуществляется объявлением одного класса дружественным (friend) по отношению к другому, которому в результате будут доступны внутренние структуры первого. Например, в листинге П.5 класс DataStore определен как производный класс от класса DataConvert для того, чтобы иметь доступ к защищенному (protected) методу convert. Вместо этого класс DataStore можно было бы сделать базовым, а внутри класса DataConvert определить его как дружественный:

```
class DataConvert
      {protected:
      int convert(char ch) {return ch-'0';}
      friend class DataStore; };
```

В этом случае функция DataConvert::convert будет видна внутри класса DataStore.

Область видимости компонента класса — это имя того класса, в котором он определен. Например, в объявлении

```
class newclass
      {protected: void setvalue(int *a, int b) { *a = b; };
      public: int x; }
```

каждый компонент можно было бы определить как newclass::setvalue и newclass::x.

При объявлении метода тело метода можно определять вне определения класса, например:

```
class newclass
      {protected: void setvalue(int *a, int b);
      public: int x; }
   newclass::setvalue(int *a, int b) { *a = b; };
```

Размещение классов. Объекты данных определенного пользователем класса объявляются так же, как и другие объекты данных; например, объект класса newclass, описанного выше, можно объявить следующим образом:

```
newclass W
```

Ссылки на компоненты этого объекта задаются так же, как и на компоненты структуры struct, например объект данных W.x, метод W.setvalue.

Указатели this. В экземплярах класса newclass, описанного выше, таких как

```
newclass i, j
```

вызов setvalue для объекта i будет определяться как i.setvalue(&m, n) и будет транслироваться так же, как если бы было записано newclass::setvalue(&i, &m, n). То есть, чтобы различать разные экземпляры класса newclass, вызывается единственная копия метода setvalue с указателем на фактический объект данных i. Аналогично вызов j.setvalue инициирует вызов newclass::setvalue(&j,&m,n). Значение этого первого указателя на фактический параметр данных называется указателем this и его можно использовать внутри объявлений методов.

Например, если определение класса выглядит следующим образом:

```
class myclass
      {methods:
private:
      myclass *Object: }
```

то компонент Object будет указателем на объект класса Myclass. Метод, который устанавливает значение указателя Object одного объекта класса Myclass таким образом, чтобы он указывал на другой объект класса Myclass, можно задать следующим образом:

```
myclass i, j:                          // Объявляются два объекта
i.setpointer(j.getpointer()):          // Указатель объекта j передается объекту i
```

где setpointer и getpointer определяются следующим образом:

```
myclass::getpointer() {return this:}
myclass::setpointer(myclass *X) {Object = X:}
```

Инициализация. Если имя метода, указанное в его объявлении, совпадает с именем класса, этот метод называется *конструктором* класса и вызывается всякий раз, когда в памяти размещается объект данного класса. При объявлении конструкторов они не должны возвращать значение или иметь возвращаемый тип. Аналогично метод с названием ~имя_класса называется *деструктором* и вызывается всякий раз, когда происходит освобождение памяти, занимаемой экземпляром класса. Например, следующий код:

```
class rational
      { public:
rational(int a: int b) {numerator = a: denominator = b: }
~rational() {numerator = 0: denominator = 0: }        // разрушение
                                                      //значения
        private:
        int numerator:
        int denominator:
        ...}
   main()
        {rational NewValue(1,2):}
```

размещает NewValue как объект типа rational с компонентами 1 и 2. Добавляя объявление

```
rational() {numerator = 1: denominator = 1:}
```

в определение нашего класса, мы тем самым реализуем инициализацию объектов класса rational по умолчанию, так что отпадает необходимость присваивать им исходные значения

```
rational x, y, z
```

Производные классы. Информацию можно передавать из одного класса в другой, используя обозначение производного класса:

```
class производный_класс: базовый_класс
        {компонент_класса1:
        компонент_класса2:
        ...
        компонент_классаn: }
```

где производный_класс — имя определяемого нового класса, а базовый_класс — это базовый класс, используемый для создания производного класса. Все открытые и защищенные компоненты базового класса становятся соответственно открытыми и защищенными компонентами в производном классе, за исключением тех имен, которые переопределяются в производном классе.

Для всех компонентов базового класса можно по умолчанию задать способ доступа, как в следующем примере:

```
class производный_класс:атрибут_доступа базовый_класс {…}
```

где атрибут_доступа может задаваться либо public, либо protected, либо private.

Виртуальные функции. В большинстве случаев связь метода и исходной программы, которую нужно выполнить, определяется статически во время трансляции. Однако это связывание можно отложить до момента выполнения программы, если использовать *виртуальные* функции. Функции в определении класса присваивается атрибут virtual, как в следующем примере:

```
virtual void setvalue(int *a, int b);
```

Если функция setvalue переопределяется в производном классе, то при каждом использовании имени setvalue для объекта производного класса будет использоваться последнее определение функции.

Если определение класса должно использоваться только в качестве базового класса, а все объекты данных определяются в производных классах, то такие классы называются *абстрактными*. Абстрактный класс можно определить как класс, содержащий чисто виртуальную функцию:

```
virtual void setvalue(int *a, int b) = 0;
```

Ни один экземпляр класса, содержащего чисто виртуальную функцию, не может быть объявлен. Производный класс, содержащий чисто виртуальную функцию, должен переопределить ее, если он будет использоваться для создания объектов этого производного класса.

Шаблоны. В языке C++ существует форма общей функции, называемая *шаблоном*. Синтаксис шаблона следующий:

```
template <class параметр_типа> class имя_класса определение
```

где определение — это либо определение класса, либо определение функции.

Шаблон имя_класса<фактический_тип> используется для описания объявления класса. Например,

```
1 #include <stream.h>
2 template <class stacktype, int size> class allstack
3     { public: stacktype stg[size]; } ;
4 allstack<int,10> x;
5 allstack<float,20> y;
6 main()
```

```
7     {x.stg[4] = 7.5; // целочисленный массив
8     y.stg[4] = 7.5;  // вещественный массив
9     cout << x.stg[4] << ' ' << y.stg[4] << endl; }
```

В строках 2–3 создается шаблон класса с именем allstack, у которого имеется *параметр типа* stacktype и целый параметр size. Этот шаблон отводит память под данные массива с именем stg, размер которого равен size, а тип — stacktype. В строках 4–5 создается объект x класса allstack, внутреннее представление которого — это целочисленный массив из 10 элементов, и объект y класса allstack, представленный вещественным массивом из 20 элементов.

В результате трансляции и выполнения этой программы получаем следующий вывод.

Сообщения компилятора об ошибках[1]:

```
sample.cc: In function 'int main ()':
sample.cc:7: warning: float or double assigned to integer data type
% execute
7 7.5
```

Предупреждение означает, что вещественное число 7.5 было преобразовано к целому числу при его присваивании переменной x.stg[4], а при присваивании переменной y.stg[4] оно оставалось вещественным.

Представление объектов в памяти

Объекты класса могут быть представлены в памяти так же, как и объекты структуры struct. Вызов метода classobject.methodname(параметр) на самом деле транслируется в процессе компиляции в следующую запись:

```
classname::methodname(&classobject. параметр)
```

Единственная сложность возникает, когда вызываются виртуальные функции, поскольку реальная функция, которую необходимо вызвать, может быть переопределена в производном классе. Для разрешения этой проблемы необходимо только сохранить таблицу переходов в объекте класса для каждой наследуемой виртуальной функции. Если она обновляется для каждой новой виртуальной функции в производном классе, то этот объект данных всегда будет указывать на текущую виртуальную функцию, которую необходимо вызвать.

П.3.2. Управление последовательностью действий

Язык C++ содержит те же операторы, что и язык C. В C++ добавлена обработка исключительных ситуаций, хотя не каждый транслятор C++ обеспечивает эту возможность.

Выражения

Выражения языка C++ те же, что и в языке C.

---

[1] Эти сообщения появятся при использовании компилятора C++ в системе UNIX, при использовании другого компилятора C++, например фирмы Microsoft для системы Windows, их не будет. — *Примеч. науч. ред.*

Операторы

Все операторы языка С можно использовать в языке С++.

Параметры функций по умолчанию. Параметры функций могут иметь значения по умолчанию, которые задаются их простой инициализацией в определениях функций. Например, функция

```
myfunction (int a; int b = 7): ...
```

может быть вызвана как myfunction(5, 12), при этом значения переменных будут a = 5 и b = 12; или как myfunction(6), тогда значения переменных будут: a = 6, b = 7. Параметры перечисляются в таком порядке, чтобы параметры, имеющие значения по умолчанию, располагались справа. (Заметим, что эта конструкция потенциально опасна, особенно если программа содержит имена перегруженных функций с различными списками параметров. Во избежание осложнений следует избегать данного способа вызова функций.)

Оператор try. Оператор try используется для обработки программно-определенных исключительных ситуаций:

```
try
      { оператор1:
      оператор2:
      ...
      if условие_исключения{throw имя_исключения}:
      }
catch(имя_исключения){ // Обработка исключительной ситуации
      }             // Конец исключительной ситуации
      }
```

Внутри любой процедуры, содержащей блок try, генерирование исключения оператором throw приводит к выполнению соответствующего оператора catch. После этого процедура завершается.

Операторы catch (которые являются определениями перегруженной функции) проверяются по очереди, пока не будет обнаружено совпадение с соответствующим оператором throw. Например,

```
throw "Это данные с ошибками."
```

может быть выявлен следующим обработчиком:

```
catch (char *ErrorString) { / *Обработка исключения */ }
```

Ввод и вывод

Входные и выходные потоки определены как стандартный набор функций. Функции cout и cin и родственные им функции описаны в следующем разделе.

Перегрузка. В языке С++ две функции могут быть перегруженными (то есть иметь одно и то же имя), если они имеют различные сигнатуры. Например,

```
int Myfunction(int a, int b) {...}: // Функция A
int Myfunction(int a) {...}:        // Функция B
```

представляют две различные функции A и B.

Если в программе дважды определено имя функции, С++ делает следующее.

1. Если сигнатуры полностью совпадают, вторая функция будет восприниматься как переопределение первой.

2. Если аргументы совпадают, а возвращаемые значения нет, то вторая функция воспринимается как ошибка.
3. Если аргументы не совпадают, функции рассматриваются как перегруженные.

Символы операций также могут быть перегружены в определении класса путем определения функции, начинающейся со слова operator, за которым следует перегруженный символ. Например, функция + для возврата первого символа строки может быть определена следующим образом:

```
#include <stream.h>
class thing
      {public: char operator+() {return stg[0]:}:
      char stg[4]: }:
main() {thing x: x.stg="abc": cout << "print x[0]: " << +x << endl:}
```

где первый аргумент — экземпляр класса thing. Данная программа осуществит следующий вывод:

```
print x[0]: a
```

Бинарный вариант операции char для + будет иметь сигнатуру

```
char operator+(thing string) ...
```

и к компонентам второго аргумента можно обращаться как string.stg. К первому аргументу можно обращаться как this → stg.

Сигнатура перегруженных операций должна быть совместима с сигнатурой их стандартных функциональных возможностей. Таким образом, + может быть унарным или бинарным, но !, например, может быть только унарным.

Перегруженная операция, такая как thing_data + int_data, должна быть определена внутри класса thing_data как функция с аргументом int_data. Таким образом, левый аргумент бинарной операции определяет класс, который содержит определение функции. В приведенном примере класс thing содержит функцию

```
operator+ (int x){...}
```

Чтобы написать функцию int_data+thing_data, необходимо иметь функцию, в которую передается аргумент класса thing, примененный к классу int. Однако тип int является встроенным. Эту проблему можно решить, если определить функцию как дружественный класс (friend) в определении класса thing:

```
friend operator+ (int x, thing y){...}
```

Однако в любом случае по крайней мере один аргумент у операции + должен быть из класса, определенного пользователем.

Неявные преобразования. Конструкция operator может быть использована для неявного преобразования типов. Пусть у нас задан объект класса thing. Конструкция

```
operator type() { return ... }
```

будет вызываться всегда, когда используется объект класса thing, а требуется объект типа type. Если добавить в определение класса thing следующую строку:

```
operator char() { return stg[0] }
```

то следующая функция будет иметь такое же действие, что и предыдущая перегруженная операция +:

```
main() {thing x: x.stg = "abc": cout << "print x[0]: " << (char) x << endl:}
```

Стандартные функции

Все функции языка С можно использовать в языке С++. Кроме того, язык С++ содержит набор стандартных потоковых функций ввода-вывода, определенных в заголовочном файле stream.h.

Функция cout << item добавляет элемент item в выходной поток. В одном операторе можно добавить несколько элементов, например, cout << $item_1$ << $item_2$ .... Для добавления символа окончания строки и печати этой строки используется специальный элемент endl. Функция cout.put поместит аргумент, состоящий из одного символа, в выходной поток.

Функция cin считывает данные в свободном формате. cin >> a >> b считывает первый элемент в переменную a, а второй в переменную b. Если требуется считать одиночный символ, это действие произведет функция cin.get (функция get аналогична функции getchar из файла stdio.h).

Функция getline(char *b, int bsize, char delim = '\n') считывает блок из не более чем bsize-1 символов, сохраняет его в массиве b и добавляет в конец null-символ \0. Параметр delim определяет, какой последний символ считывать (обычно это символ конца строки \n).

Распределение памяти. Для распределения памяти в языке С++ нет необходимости использовать системную функцию malloc. В этот язык были добавлены функции new и delete.

Функция new имя_типа размещает в памяти объект указанного типа и возвращает указатель на него. При вызове new имя_класса объект указанного класса размещается в памяти и возвращается указатель на него.

Функция delete объект освободит занятую под указанный объект память, выделенную с помощью функции new.

П.4. FORTRAN

Пример с пояснениями

В примере, приведенном в листинге П.6, происходит суммирование элементов вектора. Со времени систем пакетной обработки сохранилось правило, что числовые метки операторов записываются в позициях со 2-й по 5-ю, а символ C в первой позиции обозначает комментарий. Каждый оператор записывается на одной строке, кроме случая, если в шестой позиции следующей строки записан любой символ, что означает продолжение предыдущего оператора.

Листинг П.6. Пример суммирования массива на языке FORTRAN

```
1          PROGRAM MAIN
2             PARAMETER (MAXSIZ=99)
3             REAL A(MAXSIZ)
4   10        READ(5. 100. END=999) K
5  100        FORMAT(I5)
6                IF (K.LE.0 .OR. K.GT.MAXSIZ) STOP
7                READ *. (A(I).I=1.K)
8                PRINT *. (A(I).I=1.K)
9                PRINT *.'SUM='. SUM(A.K)
10               GO TO 10
```

```
11 999         PRINT *."All Done"
12             STOP
13             END
14 C        ПОДПРОГРАММА СУММИРОВАНИЯ
15          FUNCTION SUM(V.N)
16             REAL :: V (N) ! Новый стиль объявления
17             SUM = 0.0
18             DO 20 I = 1.N
19                SUM = SUM + V(I)
20  20            CONTINUE
21             RETURN
22             END
```

Пробелы игнорируются, поэтому для облегчения читаемости программы их можно свободно вставлять в любое место, например для обозначения вложенных блоков операторов, как показано в листинге П.6. К сожалению, многие программисты на языке FORTRAN придерживаются правила записи операторов с 7-й позиции. Однако это только руководящие принципы, и хорошие программы содержат много пустого места для улучшения читаемости. Порядковые номера строк в левой части листинга П.6 не являются частью программы, они приведены здесь только для облегчения разбора примера.

Строка 1. Имя программы MAIN.

Строка 2. MAXSIZ — определенная программистом константа. При трансляции данной программы используется значение 99, а не имя MAXSIZ.

Строка 3. Описывается одномерный массив действительных чисел размером 99, границы изменения индекса от 1 до 99. Нижняя граница полагается равной единице.

Строка 4. В этой строке происходит считывание размера массива и запись его в переменную K. Поскольку эта переменная не описана, она по умолчанию является целой. Цифра 5 после оператора READ является ссылкой на входной файл, которым является устройство стандартного ввода (например, клавиатура). Цифра 6 указывает на стандартный выходной файл (например, дисплей).

Строка 5. Здесь оператор FORMAT устанавливает, что данные будут целыми (I формат) и займут пять позиций в строке ввода. Формат F (fixed) является «фиксированным» вещественным. Например, вещественное число 5,123 могло бы иметь формат F5.3, а это означает, что для представления числа необходимо 5 символов, причем 3 крайних символа справа расположены после десятичной точки.

Строка 7. Оператор READ считывает значения элементов массива от A(1) до A(K). Вместо использования оператора FORMAT (как в операторе READ(5.101)), звездочка здесь означает использование управляемого списком оператора READ, который последовательно считывает и анализирует вещественные числа из входного потока. Переменной I присваиваются значения, которые действуют только внутри этого оператора.

Строка 10. В языке FORTRAN 77 нет конструкции while. Оператор, записанный в этой строке, передает управление назад, оператору, помеченному меткой 10, для считывания очередных данных массива. Строки 4–10 эффективно создают конструкцию

```
while не_конец_файла do
        Обработать следующий массив
        end
```

Строка 13. Конец главной программы.

Строка 14. Комментарий. Может использоваться в любом месте программы.

Строка 15–22. Подпрограмма-функция SUM. Эта функция компилируется отдельно от основной программы. Для передачи информации компилятору не используется информация из главной программы. Ошибочная строка

```
FUNCTION SUM(V.N.M)
```

также будет компилироваться, но может дать сбой, когда загрузчик попытается объединить эту подпрограмму с главной программой.

Строка 16. Хотя массив задан как V(N), в вызове функции SUM в строке 9 он ссылается на статически размещенный параметр, вещественный массив A(99) в строке 3. Здесь же показан стиль объявления, используемый в FORTRAN 90, с применением символа ::. Также FORTRAN 90 поддерживает встроенные комментарии, для чего используется восклицательный знак !.

Строка 18. В цикле DO устанавливается начальное значение счетчика I равное 1, а затем I увеличивается до N. Метка обозначает конец цикла. Если приращение не равняется 1, то требуется еще один параметр:

```
DO 20 J= 2.20.2
```

В этом цикле переменная J будет принимать все четные значения от 2 до 20.

Строка 19. Возвращаемое FORTRAN-функцией значение присваивается имени этой функции.

Строка 20. Оператор CONTINUE является пустым и используется в основном только с меткой. Здесь он заканчивает цикл DO.

П.4.1. Объекты данных

Переменные и константы. Имена переменных могут быть длиной от 1 до 31 символа, начинаются с буквы и могут содержать буквы, цифры и знак подчеркивания «_». FORTRAN нечувствителен к регистру, то есть PRINT, print, PrInT и Print обозначают одно и то же. Традиционно программы на языке FORTRAN писались с использованием верхнего регистра, но в настоящее время программисты, пишущие на FORTRAN, смешивают верхний и нижний регистры, как в большинстве других языков программирования.

Переменные не обязательно объявлять явно. Явное объявление выглядит следующим образом:

```
REAL A. B. SUM
DOUBLE PRECISION Q. R
LOGICAL :: T
```

Описание переменной T соответствует синтаксису FORTRAN 90. Если не приведено явное описание, то в силу вступает *соглашение об именах*, основанное на определении типа переменной по первой букве ее имени. В соответствии с соглашением об именах, если имя переменной начинается с букв из диапазона I-N, ее тип определяется как integer (*целая*), а все остальные переменные определяются как real (*вещественные*). Однако программист может изменить соглашение об именах, используемых в каждой подпрограмме, если начнет описание подпрограммы с оператора IMPLICIT. Например, оператор

```
IMPLICIT INTEGER(A-Z)
```

расположенный в начале подпрограммы, указывает, что все локальные переменные, не объявленные явным образом, имеют тип integer. Объявление

```
IMPLICIT NONE
```

отключает все соглашения по умолчанию в FORTRAN 90, в результате чего любая необъявленная переменная воспринимается компилятором как ошибка. Это хороший способ контроля, который следует использовать в любой программе.

Константы, определяемые программистом, можно создавать с помощью оператора PARAMETER, задаваемого в начале подпрограммы:

```
PARAMETER (KMAX=100, MIDPT=50)
```

При описании типа константы также применяется соглашение об именах либо константы можно описывать явно:

```
REAL, PARAMETER :: EPSILON = .0012
```

В языке FORTRAN используется статическая проверка типов, но она неполная. Многие возможности языка, включая параметры в вызовах подпрограмм и использование блоков COMMON, не могут быть проверены статически — отчасти потому, что подпрограммы компилируются независимо. Конструкции, которые не могут быть проверены статически, обычно остаются непроверенными во время работы программ на языке FORTRAN.

Числовые типы данных. Для *целых*, *вещественных* и *вещественных удвоенной точности* числовых типов обычно используется прямое аппаратное представление чисел. *Комплексный* тип (complex) представляется парой вещественных чисел (real), под которые отводится блок из двух машинных слов.

Для арифметических действий и преобразований между четырьмя числовыми типами имеется широкий набор элементарных операций. Основные арифметические операции (+, -, *, /) и операция возведения в степень (**) дополняются как большим набором стандартных встроенных функций, включая тригонометрические и логарифмические функции (sin, cos, tan, log), операцию извлечения квадратного корня (sqrt), нахождения максимального (max) и минимального (min) числа, так и явными функциями преобразования типов данных для различных числовых типов. Также имеются обычные операции сравнения числовых значений, которые записываются следующим образом.

Операция	Значение	Операция	Значение
.EQ.	Равно	.NE.	Не равно
.LT.	Меньше	.GT.	Больше
.LE.	Меньше или равно	.GE.	Больше или равно

Логический тип данных. Булев тип называется логическим (LOGICAL), литеральными константами этого типа являются .TRUE. и .FALSE.. Основные булевы операции представлены операциями .NOT., .AND. и .OR.. Булева эквивалентность и ее отрицание обозначаются как .EQV. и .NEQV. соответственно.

Логические выражения с числовыми данными можно конструировать обычным способом (например, (П.LT.7).OR.(B.GE.15)).

Указатели. Более старые версии FORTRAN не поддерживают указателей, поскольку все данные размещаются статически. Следовательно, построение списко-

вых структур на языке FORTRAN часто подразумевает использование больших массивов, где индекс массива служит указателем на следующий элемент списка. Такая процедура работает эффективно, если все размещаемые структуры одного типа. Однако при конструировании программ, в которых создаются произвольные структуры данных, возникают трудности. В версии FORTRAN 90 указатели добавлены как объекты нового типа данных.

Указатели объявляются следующим образом:

```
INTEGER, POINTER :: P
```

Здесь устанавливается, что P является указателем на целую переменную. *r*-значение P устанавливается равным *l*-значению целой переменной X следующим образом: P=>X.

При этом X должен быть правильным целевым объектом указателя, определенным как

```
INTEGER, TARGET :: X
```

Использование указателя в выражении равносильно использованию его *r*-значения (то есть происходит автоматическое разыменование указателя):

```
P1=>X
P2=>Y
P1=P2 ! То же, что и X = Y
```

Динамическое выделение памяти для указателя P осуществляется с помощью оператора ALLOCATE(P1), а освобождается память оператором DEALLOCATE(P1).

Структурированные типы данных

Массивы. Диапазон изменения индексов массива должен быть определен явно — либо в операторе DIMENSION (который позволяет определять тип компонентов массива через соглашение об именах), либо путем объявления массива и типа его компонентов одним из способов, описанных выше, например:

```
REAL M(20), N(-5:5)
DIMENSION I(20,20)
```

Если опущена нижняя граница индексов, она принимается равной 1. Массив I считается INTEGER (целым), если соглашение об именах для I не переопределено с помощью оператора IMPLICIT. Количество размерностей массива не должно превосходить 7.

Для выбора компонентов массива используются индексы, причем синтаксис этой операции такой же, как и синтаксис вызова функций (например, M(3) или N(I + 2)).

В отличие от большинства языков, в языке FORTRAN массивы хранятся в памяти *развернутыми по столбцам*. Это значит, что матрица сохраняется последовательно по столбцам, а не по строкам. Дескриптор не сохраняется вместе с массивом; если необходимо, его следует сохранить отдельно. Например, при доступе к двухмерному массиву A(7, 9) (смотри раздел 6.1.5) формула доступа

```
l-значение(A(7,9)) = VO + 7 × S + 9 × E
```

может быть вычислена транслятором, поскольку во время компиляции известны все границы массива и VO, S и E известны и являются константами.

Массивы, как и все данные в языке FORTRAN, размещаются статически, за исключением случаев, когда задан атрибут ALLOCATABLE.

Символьные строки. В FORTRAN можно определять такие переменные, как символьные строки фиксированной длины. Например, в FORTRAN 90 операторы

```
CHARACTER S*10, T*25
CHARACTER(LEN=7) U
```

определят S и T как символьные строки длиной 10 и 25 соответственно, а U — как символьную строку длиной 7. Объявление IMPLICIT также можно использовать для определения по умолчанию длины строки и первых букв в идентификаторе переменной типа CHARACTER, если она не объявлена явно как символьная строка. Кроме того, можно определять массивы символьных строк и функции, значениями которых являются символьные строки. Для объединения двух строк можно использовать операцию объединения (//). В операциях сравнения (.EQ., .GT. и т. д.), определенных для символьных строк, используется лексикографический порядок.

Позиции символов в строковой переменной пронумерованы от 1 до объявленной границы. С помощью следующей синтаксической конструкции из строки можно выделить подстроку:

```
Имя_строковой_переменной(позиция_первого_символа_подстроки :
   позиция_последнего_символа_подстроки)
```

Можно опустить как первый, так и последний указатель позиции символа и использовать следующие заданные по умолчанию значения:

```
позиция_первого_символа_подстроки = 1,
```

и

```
позиция_последнего_символа_подстроки = объявленная_граница_строки
```

Форматные преобразования, обеспечиваемые операциями ввода-вывода, можно также использовать для преобразования символьной строки, хранящейся в переменной или массиве, во внутреннее двоичное представление другой переменной или массива числового или логического типа (или наоборот). Используются те же операторы READ, WRITE и FORMAT, но теперь значением именованного параметра UNIT является имя переменной. Например,

```
READ (UNIT=A, FMT=200) M, N
```

определяет, что данные должны читаться из символьной строки A (как если бы переменная A была внешним файлом, преобразованным во внутреннее целое представление) и сохраняться в переменных M и N.

Определяемые пользователем типы

В версии FORTRAN 77 нет механизмов создания пользовательских типов. Все типы заранее определены языком. В FORTRAN 90 появился оператор TYPE. Например, для генерации строк переменной длины можно определить тип VSTRING:

```
TYPE VSTRING
   INTEGER :: SIZE
   CHARACTER(LEN=20):: LEN
END TYPE VSTRING
```

Объявления переменных нового типа похожи на объявления переменных других типов в FORTRAN 90:

```
TYPE(VSTRING) :: MYSTRING
```

Заметим, что символ % используется в качестве селектора поля: MYSTRING%SIZE или MYSTRING%LEN.

П.4.2. Управление последовательностью действий

Выражения

Для вычисления отдельного числа, логического значения или символьной строки можно использовать выражение. Элементарные операции ассоциативны слева направо для всех выражений, за исключением операции возведения в степень (**), которая ассоциативна справа налево. Для управления последовательностью вычислений можно использовать скобки стандартным образом. Элементарные операции имеют следующий порядок приоритетов (от высшего к низшему):

```
**
/
+ -
//
.EQ. .NE. .LT. .GT. .LE. .GE.
.NOT.
.AND.
.OR.
.EQV. .NEQV.
```

Операторы

Оператор присваивания. Синтаксис оператора присваивания имеет вид

```
переменная = выражение
```

где *r*-значение *выражения* присваивается *l*-значению *переменной*, как в следующих примерах:

```
X = Y + Z
A(1,2) = U + V - A(I,J)
```

Как *выражение*, так и *переменная* могут быть числовыми, логическими или символьными. Присваивание значения переменной типа CHARACTER корректирует длину присваиваемой строки в соответствии с объявленной длиной этой переменной, либо путем отбрасывания (если строка слишком длинная), либо путем добавления пробелов (если строка слишком короткая).

Условный оператор. Предусмотрено четыре условных оператора ветвления: традиционные конструкции IF из FORTRAN 77, а также оператор CASE языка FORTRAN 90.

Арифметический оператор IF обеспечивает трехвариантное ветвление в зависимости от того, каково значение арифметического выражения — отрицательное, нулевое или положительное:

IF (выражение) $метка_{отр_значение}$, $метка_{нулевое_значение}$, $метка_{пол_значение}$

Метки указывают, какому помеченному оператору передать управление. Эта конструкция считается устаревшей и может быть вообще удалена из следующих версий языка.

Логический оператор IF позволяет выполнить только один оператор в случае истинности логического выражения:

```
IF (выражение) оператор
```

Блочный оператор IF допускает выбор операторов if ... then ... else ... endif без использования операторных меток. Форма записи такого оператора следующая:

```
IF тестовое_выражение THEN
- последовательность операторов на отдельных строках
ELSE IF тестовое_выражение THEN
- последовательность операторов на отдельных строках
...
ELSE
последовательность операторов на отдельных строках
END IF
```

В этом операторе части ELSE и ELSE IF можно опустить. Обратите внимание на то, что из-за синтаксиса языка FORTRAN, в котором каждый оператор должен размещаться на отдельной строке, все разделители IF, ELSE IF, ELSE и END IF должны размещаться в различных строках программы.

В FORTRAN 90 добавлен оператор CASE. Его синтаксис следующий:

```
SELECT CASE выражение
CASE (вариант_значения_выражения)
   блок_операторов
... - Другие блоки CASE
CASE DEFAULT
   блок_операторов
END SELECT
```

Значение выражения должно быть целым числом, символьной строкой или логическим значением, а вариант_значения_выражения должен быть константой или диапазоном значений констант. Например, конструкция CASE(1 : 3, 5 : 6, 9) означает выбор соответствующего блока операторов, если выражение принимает значение 1, 2, 3, 5, 6 или 9. DEFAULT является необязательным; если он отсутствует, а значение выражения не удовлетворяет ни одному из перечисленных вариантов, то никакие операторы не выполняются.

Оператор цикла. Оператор цикла языка FORTRAN 90 имеет синтаксис

```
DO управление_циклом
 блок_операторов
   END DO
```

Если опущена конструкция, управляющая циклом, то получается бесконечный цикл. Выход из цикла должен осуществляться с помощью оператора EXIT.

Для цикла с фиксированным количеством повторений используется следующий синтаксис:

```
DO метка переменная = нач_знач, кон_знач, приращение
```

Здесь простая целая переменная используется в качестве счетчика повторений цикла, нач_знач — выражение, определяющее начальное значение счетчика, кон_знач определяет конечное значение счетчика, а приращение определяет число, которое должно добавляться к счетчику после выполнения каждой итерации цикла. По умолчанию приращение равно единице.

Пустой оператор. Оператор CONTINUE является пустым оператором, и его совместное использование с меткой позволяет отмечать любые места в программе. В основном его применяют, чтобы отмечать конец цикла DO или же в качестве объекта оператора GOTO. Это позволяет зафиксировать положение операторной метки, тогда как любые другие операторы можно в случае необходимости изменять.

Управление подпрограммами. Оператор CALL используется для вызова подпрограмм и задается в виде

```
CALL имя_подпрограммы(список_параметров)
```

где список_параметров — это последовательность имен фактических параметров или выражений.

Подпрограммы-функции — это подпрограммы, которые возвращают значение и могут просто использоваться в выражениях, например 2 + myfun(1.23,.TRUE.). Значение возвращается функцией простым его присваиванием имени функции перед выполнением оператора RETURN:

```
MYFUN = 27
RETURN
```

Имя функции можно использовать в качестве локальной переменной в подпрограмме. Предыдущая версия языка FORTRAN имела статическую структуру распределения памяти и не допускала рекурсии. Новая версия языка — FORTRAN 90 — поддерживает рекурсию и динамическое распределение памяти.

Выполнение программы завершается оператором STOP.

Возвращение из подпрограммы в вызывающую программу происходит с помощью оператора RETURN.

Ввод и вывод

В языке поддерживаются как последовательные файлы, так и файлы прямого доступа, а также предусмотрен широкий набор операций ввода-вывода. Для операций ввода-вывода используется девять операторов: READ, WRITE и PRINT определяют фактическое преобразование данных, операторы OPEN, CLOSE и INQUIRE позволяют установить или запросить статус, метод доступа и другие свойства файла, а операторы BACKSPASE, REWIND и ENDFILE обеспечивают установку указателя позиции файла.

Текстовый файл (последовательность символов) в языке FORTRAN называется *форматированным* файлом, остальные файлы — *неформатированными*. Во время передачи данных в форматированный файл операторы READ, WRITE и PRINT преобразуют значения данных из внутреннего представления в символьную форму. Передача данных в неформатированный файл оставляет данные в их внутреннем представлении. Ввод и вывод данных с помощью операторов READ, WRITE и PRINT в форматированный файл может осуществляться под *управлением списка*. Это означает, что явно не задается никакого определения формата, а, как видно из листинга П.6, используется внутренний, определенный системой формат. Альтернативным способом является задание программистом явного формата:

```
метка FORMAT (последовательность_спецификаций_формата)
```

Метка используется в операторах READ, WRITE и PRINT. Например, чтение последовательности целых чисел, отформатированных так, что в одной строке размещается восемь чисел, каждое из которых занимает пять позиций, можно задать с помощью следующих операторов READ и FORMAT:

```
    READ 200. N. (M(K). K=1.7)
200 FORMAT (8I5)
```

То же самое можно описать, поместив определения формата внутрь оператора READ:

```
READ "(8I5)" N. (M(K). K=1.7)
```

Этот оператор READ определяет, что первое целое число записывается в переменную N, а следующие — в компоненты 1–7 вектора M. В операторе FORMAT заданы дескриптор I преобразования в целочисленный формат, ширина поля вывода каждого числа (5) и коэффициент повторения (8), означающий, что в строке ввода располагаются восемь целых чисел, начиная с первой позиции. Для различных типов числовых и символьных данных предусматривается большое количество возможных дескрипторов форматов данных. Некоторые наиболее распространенные приведены далее (w — ширина поля ввода-вывода, d — количество десятичных цифр).

Дескриптор	Тип данных	Пример использования
Iw	Целый	I5 = 12345
Fw.d	Фиксированный (вещественный)	F6.2 = 123.56
Ew.d	Экспоненциальный (вещественный)	E10.2 = -12.34E+02 = 1234
wX	Пропуск	Пропускает w символьных позиции
Aw	Символьный	A6 = abcdef
"литерал"	Константа	"abcdef" печатается как 6 символов abcdef

Обычно безопаснее при размещении данных выравнивать их по правому краю поля. При чтении числовых данных начальные пробелы будут интерпретироваться как 0.

Каждый оператор READ и WRITE начинает обрабатывать соответственно новую строку ввода или печатать на новой строке вывода. Для каждого элемента входного или выходного списка используется следующий дескриптор оператора FORMAT, пока не будет достигнут конец списка данных. Если элементов данных больше, чем дескрипторов оператора FORMAT, тогда оператор FORMAT заново начинает перебор своих дескрипторов, но с новой строки данных.

Для задания обработчика исключений при достижении конца файла или в случае возникновения ошибки в операции ввода-вывода в операторах READ и WRITE указывается метка того оператора в подпрограмме, выполняющей ввод или вывод, которому следует передать управление при возникновении исключительной ситуации. Например,

```
READ(UNIT=2. FMT=200. END=50. ERR=90) N. (M(K). K=1.7)
```

является расширенной формой оператора READ, в котором определяется, что в случае обнаружения конца файла следует передать управление оператору с меткой 50, а в случае возникновения ошибки ввода — оператору с меткой 90. Параметр UNIT = 2 указывает, что файл нужно искать на входном устройстве, которому присвоен номер 2.

Спецификации форматов, управляющих вводом и выводом данных, трактуются как символьные строки. Во время трансляции они сохраняются в форме символьных строк и интерпретируются процедурами READ, WRITE и PRINT по мере необходимости во время выполнения программы. Спецификация формата (символьная строка) может создаваться динамически или считываться во время выполнения программы и потом использоваться для управления преобразованиями ввода-вывода. В результате получается гибкая и мощная система ввода-вывода.

П.5. JAVA

Пример с пояснениями

В листинге П.7 приведен пример программы для суммирования элементов массива на языке Java. В данном примере при вводе пользователем последовательности чисел

```
41234
```

программа выдает следующий результат:

```
1234: SUM=10
```

Этот пример похож на пример программы на языке C++, приведенный в листинге П.5, что демонстрирует сходство этих языков.

Листинг П.7. Пример суммирования элементов массива на языке Java

```
 1 import java.io.*;
 2 class DataConvert {
 3 public int convert(byte ch) {return ch-'0';}};
 4 class DataStore extends DataConvert {
 5 public void initial(int a)
 6  {ci=0;
 7  size=a;};
 8 void save(int a)
 9  {store[ci++]=a;};
10 int setprint() { ci=0; return size;};
11 int printval() { return store[ci++];};
12 int sum()
13  {int arrsum = 0;
14  for(ci=0;ci<size;ci++)arrsum=arrsum+store[ci];
15  return arrsum;};
16 private static int maxsize = 9;
17 int size; // Размер массива
18 int ci;   // Текущий индекс массива
19 int[] store = new int[maxsize];};
20 class sample {
21 public static void main(String argv[])
22 { int sz,j;
23 byte[] Line = new byte[10];
24 DataStore x = new DataStore();
25 try{
26     while((sz= System.in.read(Line)) != 0)
27         {int k = x.convert(Line[0]);
28         x.initial(k);
29         for(j=1;j<=k;j++)x.save(x.convert(Line[j]));
30         for(j=x.setprint(); j>0; j--)
31             System.out.print(x.printval());
32         System.out.print("; SUM=");
33         System.out.println(x.sum());}}
34 catch(Exception e){System.out.println("File error.");}
35 } // Конец главной программы
36 } // Конец класса sample
```

Строка 1. В этом примере будет использован стандартный пакет ввода-вывода java.io.*.

Строки 2–3. Классы языка Java похожи на классы языка C++. Тип доступа public является атрибутом функции или объекта данных и определяет их открытость.

Строка 4. Для определения производных классов в языке Java используется ключевое слово extends.

Строки 12–15. Структура функций языка Java в основном похожа на структуру функций в языках C и C++.

Строка 19. Массивы обрабатываются не так, как в языках C и C++. Массив store определяется как целый массив (int[]), а затем оператор new отводит для него требуемый объем памяти.

Строка 21. Главная программа main должна быть объявлена как public и static.

Строка 24. В этом классе осуществляется ссылка на класс DataStore через его экземпляр x, создаваемый операцией new.

Строка 25. Операторы try и catch (строка 34) необходимы для обработки ошибок ввода и вывода.

Строка 26. Метод System.in.read считывает строку ввода в байтовый массив Line.

Строки 27–33. Это тот же алгоритм, что и в программе на языке C++. Метод System.out.print записывает свои аргументы в стандартный выходной файл. Метод System.out.println добавляет в конец строки символ конца строки.

П.5.1. Объекты данных

Элементарные данные. Числовые данные могут быть 8-, 16-, 32- или 64-битовыми. Целые могут быть int и long. Также существуют 32-битовые вещественные данные (float) и 64-битовые вещественные (double), которые удовлетворяют стандарту IEEE 754. В языке Java также можно сохранять данные как byte и short, хотя для осуществления каких-либо операций над ними их требуется приводить к типу int.

Символьные данные сохраняются в формате Unicode. Это международный стандарт, который создает 16-битовые символы. Например,

```
char x = 'a'
```

создает объект x, хранящий код Unicode символа a.

Булевы данные могут иметь значения true и false. В отличие от языка C в Java булевы данные не являются целыми значениями.

Структурированные объекты данных. В языке массивы определяются явно. Например, оператор

```
Newvalues Mydata[]
```

описывает массив Mydata типа NewValues, где NewValues — определенный программистом класс объектов. Однако для фактического размещения массива необходимо использовать функцию new, как, например, в следующем примере для размещения 100-элементного массива:

```
Mydata = new Newvalues[100];
```

Для фактического размещения каждого элемента массива необходимо для каждого i выполнить оператор:

```
Mydata[i] = new Newvalues()
```

Оба эти шага можно объединить в один, например,

```
int[ ] Mydata = new int[100];
```

размещает Mydata как массив из 100 элементов.

Метод length() массивов возвращает длину массива:

```
Mydata.length()
```

В данном случае будет возвращено значение 100, как и в языке C++.

Строки – это объекты с синтаксисом, близким к синтаксису строк в языке С. Соответственно

```
String Publisher = "Prentice Hall":
```

создает строковый объект Publisher и инициализирует его строкой Prentice Hall. Операция + для строк означает их конкатенацию. Так,

```
System.out.println(Publisher + " is the publisher of this book.");
```

напечатает «Prentice Hall is the publisher of this book». Метод length() также определен для строковых объектов.

В Java нет необходимости в структуре struct, имеющейся в языке С, поскольку структура классов языка Java допускает определение сложных данных с помощью простого определения класса, содержащего несколько определенных объектов.

П.5.2. Управление последовательностью действий

Операторы. В языке Java присутствуют обычные для языка С управляющие структуры if, for и while. В этом языке нет оператора goto[1]. Выход из цикла осуществляется операторами языка С break и continue, при этом следует указать метку цикла, из которого нужно выйти:

```
OuterLoop: for(i=0. i<j. i++) {
...
if (условие выхода из цикла) {break OuterLoop:}
...
}
```

Определения классов. Определения классов в языке Java похоже на определения классов в языке C++ в отношении определения переменных экземпляра (полей) и методов для нового класса, некоторые аспекты взяты из языка Smalltalk; в частности, суперкласс нового класса задается явно, например:

```
class Newvalues extends Object {
      public double x:    /* переменная экземпляра */
      public int y:       /* переменная экземпляра */
      Newvalues() { ... } /* Конструктор для Newvalues */
      }
```

где Newvalues является субклассом (производным классом) суперкласса Object и состоит из двух полей, x (double real) и y (integer).

В языке Java используется большая библиотека предопределенных классов. java.io.* – стандартная библиотека ввода-вывода, а java.awt.* – стандартная библиотека апплетов, используемая для выполнения апплетов Java, встроенных в web-страницы HTML.

[1] Но зарезервированное ключевое слово goto осталось! – *Примеч. науч. ред.*

Как уже говорилось, для размещения экземпляров класса (объектов) используются С-подобные объявления. Так,

```
Newvalues Mydata[]
```

создает массив Mydata (пустой), каждый из элементов которого имеет тип Newvalues.

Как и в языках С и С++, используется точечная нотация для ссылок на компоненты объекта Mydata: Mydata[17].x и Mydata[17].y.

Кроме того, так же как и в С++, можно определить метод Newvalues() для явного присваивания переменным экземпляра (полям) x и y значений при создании экземпляра класса.

Конструктору можно передать параметры, например:

```
Newvalues(double a: int b) {
       this.x = a:
       this.y = b:}
```

При освобождении памяти нужно вызывать метод finalize() (который действует подобно деструктору класса языка С++ ~имя_класса).

Так же как и в языке С++, в Java можно определять другие методы в описании класса и затем вызывать их с использованием точечной нотации.

В языке Java отсутствует явный вызов функции, так как в этом нет необходимости. Доступ к методу (как в System.out.println(string)) является неявным вызовом функции.

Определение производных классов. Язык Java имеет такую же структуру производных классов, как и язык С++. Newclass является производным классом класса OlderClass, если он определен следующим образом:

```
class Newclass extends Olderclass {...}
```

Newclass наследует все переменные экземпляра (поля) класса Olderclass, а вызов метода класса Newclass (например, Newclass.functionX) передается Olderclass.functionX, если метод functionX не определен в классе NewClass.

Управление доступом. Переменные экземпляра в языке Java могут быть открытыми (public), закрытыми (private) и защищенными (protected). Открытые имена доступны отовсюду, закрытые имена доступны только внутри определения класса, а защищенные имена доступны в производных классах данного класса.

Если не используется ни один из приведенных атрибутов имен, полагают, что управление доступом является *дружественным*, то есть имена являются доступными для любого определения класса внутри пакета, к которому принадлежит данный класс, но больше ниоткуда (некая форма полуоткрытого доступа).

Переменные в определении класса обычно являются переменными экземпляра — это означает, что они присутствуют в любом экземпляре объекта этого класса. Для описания переменной, общей для всех экземпляров данного класса, используется атрибут static. Например, если бы вы создавали структуру связанного списка, каждый связанный список был бы классом LinkedObject. В нем было бы объявление

```
static FreeListType FreeList:
```

которое было бы общим для всех экземпляров класса LinkedObject, чтобы управлять размещением новых элементов списка из общего списка. Методы, общие для

всех экземпляров класса и имеющие доступ только к переменным класса, также могут быть объявлены с атрибутом static.

Виртуальные методы языка C++ в языке Java называются *абстрактными* и определяются так же, как и виртуальные методы языка C++:

```
abstract void MethodName(); /* метод с пустым телом */
```

Потоки. Через потоки Java позволяет реализовать некоторый уровень многопроцессорной обработки. Например, браузер HotJava параллельно с браузером, отображающим HTML-файлы пользователя, запускает программу сборки мусора. Это позволяет избежать остановки отображения файлов на время ожидания завершения процесса сборки мусора. Также и метод для отображения графики может быть отдельным потоком по отношению к методу отображения текста, позволяя реализовать более быструю передачу полезной информации.

Потоки запускаются командой new Thread(объект). Атрибут synchronized (synchronized void startSort()) в определениях нескольких методов запрещает одновременное выполнение двух (и более) таких методов, тем самым предотвращая тупиковые ситуации, например одновременный доступ к совместно используемым данным.

Стандартные функции

Язык Java был создан для облегчения создания web-страниц. Поэтому он включает в себя библиотеку классов Abstract Window Toolkit (AWT) для построения пользовательских интерфейсов. В ней предусмотрены средства для создания фреймов, отображения их на экране, возможности добавления меню, полос прокрутки и текста. Следующие методы дают только небольшое представление о некоторых из имеющихся функциональных возможностей, сама же библиотека классов настолько огромна, что нет возможности полностью описать ее в данной книге.

Библиотека AWT создает иерархию классов. На верхнем уровне находятся графика, компоненты (кнопки, контейнеры — например, фреймы, — текстовые области) и границы. Для создания сложных отображаемых структур используются различные производные классы. Объект Window (окно) не имеет ни границ, ни полосы меню. Он может генерировать события WindowOpend или WindowClosed, которые могут использоваться для создания экземпляра окна. Фреймы являются окнами с заголовком и границей, для которых определено больше событий, чем для простого окна.

Метод show() отображает на экране фрейм (субкласс окна).

Конструктор Frame("имя_фрейма") создает новый фрейм с указанным именем.

Метод setSize(M, N) устанавливает размер фрейма M × N пикселов.

Метод MenuBar() создает во фрейме строку меню.

Метод MenuItem("имя") создает элемент меню с указанным именем.

Метод Button("имя") создает кнопку с указанным именем.

Метод add(объект) добавляет указанный объект (например, меню или кнопку) во фрейм.

Метод drawstring("текст", X, Y) помещает указанный текст во фрейм в точке с координатами (X, Y) в пикселах.

П.6. LISP

Пример с пояснениями

В примере, приведенном в листинге П.8, снова происходит суммирование элементов массива. В данном случае мы задаем функции прямо в сеансе LISP. Хотя используемый нами LISP имеет функцию (load "имя_файла"), которая считывает определения функций языка LISP из файла. Символ > обозначает приглашение LISP вводить команды.

Листинг П.8. Сумма элементов числового вектора на языке LISP

```
 1 %lisp
 2 >: Сохранение значений в виде списка символов
 3 >(define (SumNext V)
 4    (cond ((null V) (progn (print "Sum=") 0))
 5    (T (+ (SumNext (cdr V)) (car V)) ) ))
 6 SUMNEXT
 7 >: Создание вектора из введенных значений
 8 (defun GetInput(f c)
 9    (cond ((eq c 0) nil)
10    (T (cons (read f) (GetInput f (- c 1))))))
11 GETINPUT
12 >(defun DoIt()
13    (progn
14    (setq infile (open '"lisp.data"))
15    (setq array (GetInput infile (read infile)))
16    (print array)
17    (print (SumNext array))))
18 DOIT
19 >(DoIt)
20
21 (1 2 3 4)
22 "Sum="
23 10
24 10
```

Строка 1. Вызов интерпретатора LISP.

Строка 2. Комментарии в языке LISP начинаются с символа «:» и продолжаются до конца строки.

Строка 3. Функция SumNext объявлена как функция с одним параметром. Она возвращает сумму элементов, содержащихся в ее параметре. Определением функции SumNext является выражение, записанное в строках 4–5.

Строка 4. Функция cond ищет первую истинную пару < предикат, выражение >. Первая пара имеет предикат null V, который принимает истинное значение после обработки последнего введенного значения. Функция печатает SUM= и возвращает управление.

Конструкция progn последовательно выполняет последовательность выражений, в данном случае — функцию print, а за ней — выражение 0 как значение по умолчанию.

Строка 5. Если вектор V не пустой (обычный случай), предикат T становится вариантом по умолчанию для функции cond в строке 4. Вектор V без первого элемента передается рекурсивно функции SumNext, и после возвращения голова V прибавляется к ранее вычисленной сумме.

Строка 6. LISP печатает имя функции SUMNEXT как результат выполнения строк 3–5.

Строки 7–11. В этих строках определяется функция GetInput, имеющая два параметра: указатель входного файла f и количество элементов вектора, которое осталось прочитать, c. Это определение демонстрирует синтаксис функции defun, используемой для определения функции; в Scheme применяется другая конструкция, define. Функция GetInput возвращает список значений прочитанных данных.

Строка 9. Функция cond сначала сравнивает счетчик c с нулем, и если он равен нулю, возвращает пустой список.

Строка 10. Если счетчик не равен нулю, функция GetInput создает новый вектор, читая атом из файла f и добавляя его в голову списка, возвращаемого после рекурсивного вызова GetInput с параметрами f и счетчиком, уменьшенным на единицу.

Строки 12–18. Здесь определяется основная управляющая функция программы. Функция DoIt не имеет параметров.

Строка 13. Функция progn последовательно выполняет выражения строк 14–17.

Строка 14. Функция open открывает файл lisp.data и возвращает указатель на файл. Функция setq присваивает этот указатель переменной infile. Последовательность действий следующая: вычисляется аргумент open, а затем осуществляется присваивание переменной infile.

Строка 15. Вызывается функция GetInput (строки 8–10) с параметром infile, который был определен в строке 14, и результатом выполнения функции read, который будет счетчиком количества элементов вектора, подлежащих суммированию. Функция setq присваивает эти входные данные переменной array.

Строка 17. Входные данные (список array) передаются функции SumNext, определенной в строках 3–5, которая возвращает сумму членов списка.

Строка 19. Вызывается функция DoIt.

Строки 20–24. Если файл lisp.data содержит следующие данные:

```
4 1 2 3 4
```

то в строках 20–23 представлен результат работы программы. В этом примере дважды напечатано число 10 — один раз как результат выполнения функции print в строке 17, второй раз как значение, возвращенное функцией DoIt.

П.6.1. Объекты данных

Элементарные типы данных

Элементарными типами объектов данных в языке LISP являются списки и атомы. Определения функций и списки свойств являются специальными типами списков особой важности. Во всех реализациях обычно также предусмотрены массивы, числа и строки, однако эти типы играют не столь важную роль в языке.

Переменные и константы. В языке LISP *атом* является основным элементарным типом объектов данных. Атом иногда называют *литеральным атомом* (*literal atom*), чтобы отличить от *числа* (*numeric atom*), которое большинством функций языка LISP также классифицируется как атом. Синтаксически атом является идентификатором — это последовательность букв и цифр, начинающаяся с буквы. Для

идентификаторов обычно не учитывается регистр задания букв. В определениях функций языка LISP атомы выполняют обычную роль идентификаторов — они используются в качестве имен переменных, функций, формальных параметров и т. д.

Однако во время выполнения атомы языка LISP являются не просто идентификаторами. Любой атом — это сложный объект данных, представленный областью памяти, которая содержит дескриптор типа для атома вместе с указателем на *список свойств*. Список свойств содержит различные свойства, ассоциированные с атомом, одним из которых всегда является его *печатное имя* — символьная строка, представляющая атом при вводе и выводе. Другие свойства представляют собой различные связывания для атома, которые могут включать функцию, с именем, соответствующим имени атома, и другие свойства, назначаемые программой во время выполнения.

Когда бы атом ни появлялся в качестве компонента другого объекта данных, например списка, он представляется указателем на область памяти, служащим представлением этого атома во время выполнения. Таким образом, любая ссылка на атом ABC во время выполнения программы LISP представлена указателем на эту область памяти.

Каждый атом также обыкновенно появляется как компонент в определяемой системой центральной таблице, называемой *списком объектов* (*ob_list*). Таблица *ob_list* обычно организуется как хэш-таблица, которая позволяет осуществлять эффективный поиск печатного имени (символьной строки) и получать значение указателя на атом с этим печатным именем. Например, когда вводится список и очередной элемент является символьной строкой "ABC", представляющей атом ABC, функция read ищет в таблице *ob_list* элемент "ABC", который также содержит указатель на область памяти, где хранится атом ABC. Этот указатель вставляется в список, конструируемый в соответствующей точке.

Можно использовать *числа* (числовые атомы) в целом формате или формате с плавающей точкой. Используется аппаратное представление, но требуется также и дескриптор времени выполнения, так что для каждого числа используется два слова. Однако это представление хорошо сочетается с представлением, используемым для литеральных атомов; число — это атом, обозначенный как принадлежащий специальному типу, с указателем на битовую строку, представляющую число, вместо указателя на список свойств.

Строки в языке LISP представляются обычными строками символов. Следует знать, что одиночная кавычка (') интерпретируется как функция quote для литеральных (не вычисляемых) аргументов функции.

Список свойств. Каждый литеральный атом имеет связанный с ним список свойств, доступ к которому осуществляется через указатель, хранимый в области памяти, представляющей атом. Список свойств является обычным списком языка LISP, отличающийся только тем, что его элементы располагаются в чередующейся последовательности логическими парами имя свойства/значение свойства. Если атом является именем функции, его список свойств содержит имя свойства, дающее тип функции, и указатель на список, представляющий собой определение функции. Программист может добавлять другие требуемые свойства, некоторые элементарные операции также могут добавлять свойства.

Структурированные типы данных

В большинстве реализаций языка LISP предусмотрены некоторые разновидности таких объектов данных, как векторы или массивы. Однако в разных реализациях эти объекты представлены весьма неодинаково, что объясняется сравнительно небольшой их значимостью для написания программ на языке LISP. Типичная реализация содержит функцию mkvect(bound), которая создает векторный объект данных с диапазоном изменения индексов от 0 до заданной границы bound. Каждый компонент вектора может содержать указатель на любой объект данных LISP; изначально значения указателей установлены равными nil. Функция getv(vector. subscript) возвращает указатель, хранящийся в указанном компоненте вектора-аргумента. Таким образом, функция getv является LISP-версией операции индексации для получения значения компонента вектора. Функция putv используется для присвоения нового значения компоненту вектора.

Определяемые программистом функции пишутся в форме списков для ввода, как видно из листинга П.8. Программы ввода в языке LISP не делают различий между определениями функций и списками данных, а просто преобразуют все списки во внутреннее представление связанных списков. Каждый встречающийся атом ищется в таблице *ob_list*, и если атом уже существует, то возвращается указатель на него, если же атома в списке нет, то создается новый атом.

Инициализация и присваивание. В отличие от других языков прямое присваивание не играет важной роли в программировании на языке LISP. Многие программы на языке LISP написаны вообще без операций присваивания, а для достижения того же эффекта неявным образом используются рекурсия и передача параметров. Однако присваивание используется внутри сегментов prog, где программа на языке LISP принимает вид обычной последовательности операторов. Основной операцией присваивания является setq. Выражение (setq x val) присваивает переменной x новое значение val, причем результатом этого выражения является величина val (таким образом, setq — это функция, но ее значение обычно игнорируется). Элементарная операция set аналогична setq, за исключением того, что переменная (а именно атом), которой присваивается значение, может быть вычислена. Например, (set(car L) val) эквивалентно (setq x val), если атом x является первым элементом списка L. Функции rplaca и rplacd позволяют присваивать значения соответственно полям car и cdr любого элемента списка. Например, (rplaca L val) присвоит значение val в качестве первого элемента списка L, заместив текущий первый элемент.

П.6.2. Управление последовательностью действий

LISP не делает различий между данными LISP и программами LISP. Этим объясняется простота интерпретатора LISP. По сравнению с другими языками, которые рассматриваются в данной книге, выполнение программы на языке LISP интересно тем, что основной интерпретатор может быть описан на LISP. Функции apply передается указатель на функцию, которую следует выполнить, и apply интерпретирует определение этой функции, используя вторую функцию eval для вычисления каждого аргумента.

Транслятор языка LISP — это просто функция read. Функция read сканирует символьную строку из входного файла или с терминала, ища правильный идентификатор или списковую структуру. Если найден одиночный идентификатор, функция просматривает таблицу *ob_list* для получения указателя на соответствующий атом (или для создания нового атома, если не найден атом с заданным печатным именем). Если найдена списковая структура, начинающаяся с символа «(», то сканируется каждый элемент списка, пока не будет найден соответствующий символ «(». Каждый элемент списка транслируется во внутреннюю форму, и в соответствующем месте в список вставляется указатель car. Числа транслируются в их двоичный эквивалент с дескриптором в отдельном слове. Если найден подсписок, начинающийся с другого символа «(», то рекурсивно вызывается функция read для построения внутреннего представления подсписка. Указатель на подсписок вставляется как указатель car следующего компонента основного списка. Заметим, что этот процесс трансляции не зависит от того, является ли список определением функции или списком данных; все списки обрабатываются одинаково.

Выполнение программы на языке LISP состоит из вычислений функций LISP. Последовательность выполнения программы обычно задается последовательностью вызовов функций (может быть, рекурсивных) и условными выражениями.

Выражения

Программа на языке LISP состоит из последовательности определений функций, где каждая функция является выражением в кембриджской польской записи.

Условные выражения. Основной структурой для обеспечения альтернативных выполняемых последовательностей в программе на языке LISP является условное выражение. Его синтаксис следующий:

```
(cond вариант1
      вариант2
      ...
      вариантn
      (T выражение_по_умолчанию))
```

где каждый из перечисленных вариантов — это пара(предикат$_i$. выражение$_i$).

Функция cond выполняется путем последовательного вычисления каждого предиката$_i$, и вычисления выражения$_i$ для первого предиката, вернувшего значение истина (T). Если все предикаты ложны, то выполняется выражение_по_умолчанию.

Операции над атомами. Обычно операции над атомами включают: тестовую функцию atom, которая распознает указатель на слово списка и указатель на атом (путем проверки дескриптора слова), функцию numberp, которая проверяет, является ли атом литеральным или числовым, функцию eq, которая проверяет, являются ли два литеральных атома одним и тем же атомом (простой проверкой того, указывают ли оба ее аргумента на одну и ту же область памяти), функцию gensym, генерирующую новый атом (и не помещающую его в таблицу *ob_list*), функцию intern, помещающую атом в таблицу *ob_list*, и функцию remob, удаляющую атом из таблицы *ob_list*.

Язык LISP содержит основные арифметические операции: +, -, *, / — и некоторые другие. Их синтаксис такой же, как и у других операций языка LISP. Так, $A + B * C$ записывается как (+ A (* B C)). Все арифметические операции являют-

ся общими операциями и принимают аргументы как целого, так и вещественного типа и при необходимости осуществляют преобразование типов.

Операция сравнения zerop проверяет, не равен ли нулю ее аргумент, операции greaterp и lessp представляют собой стандартные операции сравнения *больше чем* и *меньше чем*. Результатом этих операций является либо атом nil, представляющий ложь, либо атом T, представляющий истину (любое значение, не равное nil, обычно представляет истину в языке LISP).

В языке LISP нет булева типа. Булевы операции and, or и not представлены в форме функций. Операция and обрабатывает произвольный список не вычисленных аргументов, вычисляя их все по очереди и возвращая значение nil, если один из ее аргументов оказывается равным nil. Операция or работает подобным же образом.

Операции над списками. Элементарные операции car и cdr возвращают, соответственно указатель car и указатель cdr заданного элемента списка. Если список L является операндом этих операций, то (car L) возвращает указатель на первый элемент этого списка, а (cdr L) возвращает указатель на список, из которого удален первый элемент.

Элементарная операция cons получает в качестве операндов два указателя, отводит для нового элемента списка слово памяти, сохраняет два указателя в car- и cdr-полях слова и возвращает указатель на новое слово. Если второй операнд является списком, то первый элемент добавляется в голову списка и возвращает указатель на расширенный список.

Элементарные операции car, cdr и cons являются основными операциями для выборки компонентов списка и создания списков. С помощью операции cons можно элемент за элементом создать любой список. Например,

```
(cons A (cons B (cons C nil)))
```

создает список из трех элементов, на которые ссылаются как на A, B и C. Аналогично, если L = (A B C) — список, то (car L) возвращает A, (car(cdr L)) — B, (car(cdr(cdr L))) — C. Используя операции car, cdr и cons, любой список можно разбить на составные элементы и создать новый список из этих или других элементов.

Элементарную операцию list можно использовать в качестве замены длинных последовательностей операций cons. Операция list получает любое количество аргументов и конструирует из них список, возвращая указатель на получившийся список.

Операция quote позволяет записать в программе любой список или атом в виде литерала, как описано в разделе 6.1.7.

Элементарная операция replaca используется для замены поля указателя car в слове списка на новый указатель; replacd используется для замены указателя cdr. Эти две операции следует использовать осторожно, поскольку они действительно изменяют содержимое слова списка, а также возвращают значение, следовательно, имеют *побочные эффекты*. Из-за сложного способа связывания списков в LISP эти побочные эффекты могут случайно повлиять и на другие списки помимо того, который модифицируется.

Операция null проверяет, пуст ли данный список (равняется ли атому nil), операцию append можно использовать для соединения двух списков, а операция equal сравнивает два списка на равенство соответствующих элементов (вызывая себя рекурсивно для соответствующих пар элементов).

Операции над списками свойств. Основные функции обеспечивают вставку, удаление и доступ к свойствам списков свойств. Функция put используется для добавления пары имя свойства/значение свойства в список свойств. Функция get возвращает текущее значение, ассоциированное с заданным именем свойства. Функция remprop удаляет пару «имя–значение» из списка свойств. Например, для добавления пары «имя–значение» (age, 40) в список свойств атома mary следует записать:

```
(put 'mary 'age 40)
```

Далее в программе значение свойства age атома mary можно получить через вызов функции

```
(get 'mary 'age)
```

Удаление свойства age атома mary выполняется с помощью вызова:

```
(remprop 'mary 'age)
```

Элементарные операции, которые определяют функции, также могут изменять списки свойств для специальных имен свойств, таких как expr или fexpr (см. далее).

Ввод и вывод

Ввод и вывод осуществляется просто через вызов функций. Функция read считывает следующий атом с клавиатуры. В строках 14–15 листинга П.8 показано, как можно открыть файл данных с помощью функции open и прочитать его содержимое с помощью вызова (read указатель_файла).

Вызов (print обьект) распечатает указанный объект в читаемом формате.

Определение функций

Функция defun используется для создания новых функций. Она имеет следующий синтаксис

```
(defun имя_функции(аргументы)выражение)
```

где аргументы являются формальными параметрами функции, а телом функции — единственное выражение (которое может быть последовательностью progn или другими сложными выполняемыми последовательностями).

В Scheme для определения функций можно использовать синтаксис

```
(define (имя_функции аргументы) выражение)
```

В начале 60-х гг., когда был разработан первый вариант языка LISP, определение функций основывалось на λ-выражениях, которые мы обсуждали в разделе 4.2.2. λ-выражения можно использовать для определения функций

```
(define имя_функции (lambda(параметры)(тело)))
```

где имя функции и имя каждого параметра — атомы, а тело может быть любым выражением, включающим элементарные операции или определенные программистом функции.

Действие объявления функции достаточно просто. Определения интерпретируемых функций обычно хранятся как пары «атрибут–значение» в списке свойств атома, представляющего имя функции. Для обычных функций, таких как определенные ранее, используется имя атрибута expr, а связанное с ним значение являет-

ся указателем на списочную структуру, представляющую определение функции. Таким образом, выполнение ранее описанной операции define эквивалентно выполнению следующей операции:

```
(put 'имя_функции 'expr '(lambda(параметры)(тело)))
```

Многие системы LISP отводят специальную область в атоме (заголовочный блок) для определения функции, названной по этому атому, чтобы избежать необходимости поиска определения функции в списке свойств каждый раз, когда она вызывается. Специальная область памяти отведена для обозначения типа функции (expr, fexpr и т. д.) и содержит указатель на списочную структуру, представляющую определение функции. В таких реализациях обычно используются специальные элементарные операции для прямого получения, вставки и удаления определений функций из специальной области.

Стандартные функции

Многие из перечисленных далее функций существуют в большинстве реализаций LISP. Однако прежде, чем их использовать, лучше ознакомиться с руководством по используемой версии LISP.

Функции работы со списками:

- (car L) возвращает первый элемент списка L. (caar L) = (car(car L)), и т. д.
- (cdr L) возвращает список L без первого элемента. (cddr L) = (cdr(cdr L)), и т. д.

Функции car и cdr могут использоваться совместно в произвольном порядке, например, (caddr L) =(car(cdr(cdr L))).

- (cons x y) возвращает список L такой, что (car L) = x, а (cdr L) = y.
- (list x y z) возвращает список (x y z).
- (quote x)(or 'x) не вычисляет x.

Предикаты:

- (atom x) возвращает истину, если x — атом.
- (numberp x) возвращает истину, если x — число.
- (greaterp x y) возвращает истину, если x > y.
- (lessp x y) возвращает истину, если x < y.
- (null x) возвращает истину, если x — пустой атом nil.
- (and x y) возвращает x ∧ y.
- (or x y) возвращает x ∨ y.
- (not x) возвращает ~ x.
- (eq x y) возвращает истину, если x и y — одинаковые атомы или списки.
- (equal x y) возвращает истину, если x и y — списки с одинаковыми элементами.

Арифметические функции:

- (+ x y) возвращает x + y для атомов x и y. Аналогично работают *, / и -.
- (rem x y) возвращает остаток от деления x/y.

Функции ввода и вывода. Многие из этих функций различаются в различных реализациях LISP.

- (`load имя_файла`) считывает файл с указанным именем как последовательность определений языка LISP. Это основное средство загрузки программ из файлов.
- (`print x`) печатает элемент `x`. В Scheme эта функция называется также `display`.
- (`open имя_файла`) открывает файл с указанным именем и возвращает указатель на этот файл. Обычный способ использования этой функции — сохранение указателя на данный файл с помощью функции `setq`:

```
(setq infile (open 'имя_файла))
```

  и использование переменной `infile` в операторе `read`.
- (`read`) считывает с терминала следующий атомарный символ (число, символ или строку). (`read указатель_файла`) считывает из файла, заданного своим указателем, следующий атомарный символ, если данный файл был предварительно открыт.
- (`help`) или (`help 'command`) может предоставить полезную информацию.
- (`trace имя_функции`) осуществляет трассировку выполнения указанной функции, используется для облегчения выявления ошибок программы.
- (`bye`) осуществляет выход из LISP.

Абстракции и инкапсуляция

Базовый LISP не предусматривает возможности абстракции данных. Однако Common LISP содержит объектную систему Common LISP Object System (CLOS). CLOS стал результатом слияния четырех объектно-ориентированных расширений LISP в середине 80- гг.: New Flavors, CommonLoops, Object LISP и Common Object [106]. CLOS предоставляет следующие возможности:

1. Множественное наследование, которое обеспечивается наследованием `mixin` (раздел 7.2.3).
2. Общие функции.
3. Метаклассы и метаобъекты.
4. Способ создания и инициализации объектов, который допускает пользовательское управление процессом.

П.7. ML

Пример с пояснениями

В листинге П.9 представлен пример суммирования элементов массива. Язык ML работает как интерпретатор, в этом примере используется Standard ML, разработанный в Нью-Джерси.

Листинг П.9. Суммирование элементов массива на языке ML

```
1 %editor prog.sml
2 fun digit(c:string):int = ord(c)-ord("0");
3 (* Значения сохраняются в виде списка символов *)
4 fun SumNext(V) = if V=[ ] then (print("\n Sum="); 0)
```

```
 5             else (print(hd(V));
 6             SumNext(tl(V))+digit(hd(V)));
 7 fun SumValues(x:string):int= SumNext(explode(x));
 8 fun ProcessData() =
 9     (let val infile = open_in("data.sml");
10        val count = digit(input(infile,1))
11     in
12        print(SumValues(input(infile,count)))
13     end;
14     print("\n"));
15 %editor data.sml
16 41234
17 %sml
18 - use "prog.sml";
19 [opening prog.sml]
20 val digit = fn : string -> int
21 val SumNext = fn : string list -> int
22 val SumValues = fn : string -> int
23 val ProcessData = fn : unit -> unit
24 val it = () : unit
25 - ProcessData();
26 1234
27 Sum=10
28 val it = () : unit
```

Строки 1–14. Для создания программы вызывается редактор. Текст программы может быть напечатан прямо в сеансе работы с ML, а не подключаться с помощью команды use, как в строке 18.

Строка 2. Функции перед их использованием необходимо объявлять. Функция digit получает в качестве аргумента один символ и возвращает целое значение, преобразуя полученный символ в целое число, как и в вышеописанном примере на языке С из раздела П.2. Символ «1» на единицу больше, чем символ «0», «2» больше, чем «0», на 2 и т. д. Функция ord возвращает целое значение первого символа своего строкового аргумента.

Строка 3. Так выглядят комментарии на языке ML.

Строки 4–6. Здесь определяется функция, которая рекурсивно выбирает очередную начальную цифру из массива и добавляет ее значение к сумме.

Строка 5. Если список не пустой, то должно вычисляться следующее значение. Печатается заголовок списка. Строки 5–6 формируют последовательность из двух выражений, которые нужно вычислять как часть выражения else.

Строка 7. Здесь инициализируется сумма элементов списка. Входная строка преобразуется в список символов с помощью встроенной функции explode, затем функция SumNext вычисляет значение каждого символа из списка.

Строка 9. Прежде чем вычислять выражение в строке 12, вычисляются выражения let в строках 9 и 10. infile инициализируется как указатель на файл данных. Побочным эффектом вычисления этого выражения является открытие файла данных.

Строка 12. Переменная count содержит количество символов, которые следует прочитать. Это количество символов считывается, и результирующая строка передается для обработки в функцию SumValues. Получившееся значение выводится на печать.

Строка 14. Печатается символ конца строки.

Строки 15–16. Для создания файла данных вызывается редактор.

Строки 17–18. Загружается Standard ML, и считывается файл prog.sml, содержащий последовательность определений функций.

Строки 19–24. Вывод интерпретатора ML, указывающий, что он обрабатывал определения функций из входного файла. Печатаются сигнатуры функций digit, SumNext, SumValues и ProcessData.

Строка 25. Вызывается функция ProcessData().

Строки 26–28. Приведенные данные (в том числе вычисленная сумма) — результат работы программы. Выход из ML осуществляется с помощью индикатора конца файла — обычно это сочетание клавиш <Ctrl+D>, введенное с клавиатуры.

П.7.1. Объекты данных

Элементарные типы данных

Переменные и константы. В языке ML существуют целые (int) константы (например, 1, 2, 3, 4), вещественные числа (real) (например, 1.2, 3.4, 10.E3) с записью в общепринятой нотации, булевы значения (bool) (то есть истина или ложь) и строки (string) (например, "1", "abc"). Как для вещественных, так и для целых отрицательных чисел используется символ «~» (тильда). В языке ML регистры различаются, так что True и ABC — не то же самое, что true и abc. Только true и false являются булевыми литеральными константами.

В ML используются соглашения языка С для представления управляющих символов. Чтобы встроить управляющий символ в строку, следует использовать последовательность символов \n для символа новой строки, \" — для кавычек (без экранирующего символа обратной наклонной черты кавычки в строке воспринимались бы как признак ее завершения), \t для символа табуляции и \ddd, где ddd — трехзначный восьмеричный код символа.

Идентификаторы — это последовательности символов, начинающиеся с буквы или символа апострофа (') и содержащие буквы, цифры и символ подчеркивания (_). Идентификаторы, начинающиеся с апострофа, рассматриваются как идентификаторы типов.

Структурированные типы данных

В языке ML существуют следующие структурированные объекты: кортежи, списки и записи.

Кортеж (tuple) представляет собой последовательность объектов, разделенных запятыми и заключенных в скобки. (12, "abc", 3) — это кортеж типа int*string*int. Кортеж может иметь неограниченное количество уровней вложенности. Кортеж

```
(2, "a", (3, 4.3), "xyz", (true, 7))
```

имеет тип (int*string*(int*real)*string*(bool*int)).

К i-му элементу кортежа можно обратиться с помощью конструкции #i. Например, второй элемент кортежа определяется как #2, например #2(2, 4, 6) = 4.

Список (list) является кортежем объектов одного типа. Четыре строки в списке можно было бы задать как ["a", "b", "c", "d"], а результирующий тип обозна-

чается как string list. Аналогично, [[1,2], [3,4]] будет типа int list list или список списков целых величин.

Функция nil обозначает пустой список.

Для задания *записи* (record) в языке ML используется синтаксис {$label_1$ = значение, $label_1$ = значение, ...}, где компоненты записи выбираются операцией #label. На самом деле кортежи являются специфической разновидностью записи. Кортеж (10, 23, "a") является экземпляром записи {1=10, 2=23, 3="a"}.

Инициализация. Чисто аппликативные языки не имеют концепции присваивания значения области памяти. Однако в языке ML значение может быть связано с идентификатором с помощью конструкции val. Так,

```
val идентификатор = выражение
```

связывает значение выражения с именем идентификатора. Например, val X = 7 выведет на печать строку

```
val X = 7 : int
```

означающую, что X имеет значение 7 типа int. В языке ML также существует ссылочный тип, который более тесно связан с хранением данных. Он будет описан позже.

Заметим, что символ «=» обозначает не присваивание, а операцию унификации, которая предписывает ассоциировать новую копию идентификатора X со значением 7. Это можно продемонстрировать, опустив ключевое слово val. В ответ на ввод X = 8 ML выведет на печать строку

```
val it = false : bool
```

означающую, что унификация текущего значения X (то есть 7) со значением 8 ложна и поэтому возвращается значение false.

Определяемые пользователем типы

Пользовательские типы можно определить с помощью оператора datatype:

```
datatype идентификатор = выражение_типа
```

Например, тип direction (направление) можно определить так:

```
- datatype direction = north | east | south | west;
      con east : direction
      con north : direction
      con south : direction
      con west : direction
```

что создает четыре константы типа direction. Мы создали тип, похожий на перечисляемый тип в C и Pascal. Оператор

```
- val NewDirection = north;
```

устанавливает тип NewDirection как тип direction со значением north.

Это определение пользовательских типов можно распространить на типы данных с древовидной структурой. Например,

```
- datatype Mydirection = dir of direction;
      datatype Mydirection
      con dir : direction -> Mydirection
```

создает тип Mydirection, состоящий из функции выбора dir.

Объекты типа Mydirection задаются как кортежи (то есть dir(north)), например:

```
- val Heading = dir(north);
val Heading = dir north : Mydirection
```

Без кортежа получим значение направления:

```
- val NewHeading = north;
val NewHeading = north : direction
```

Компоненты типов данных похожи на вызовы функций:

```
- val NewDirection = dir(NewHeading);
val NewDirection = dir north : Mydirection
```

Сила типов данных заключена в возможности создавать произвольные древовидные структуры данных. Например, на рис. П.2 изображено дерево, состоящее из узлов NumEntry и CharEntry. Спецификация будет иметь следующий вид:

```
-   datatype Tree = Empty |
        NumEntry of int * Tree * Tree |
        Cha Entry of string * Tree;
datatype Tree
        con Char Entry : string * Tree → Tree
        con Empty : Tree
        con NumEntry : int * Tree * Tree → Tree
```

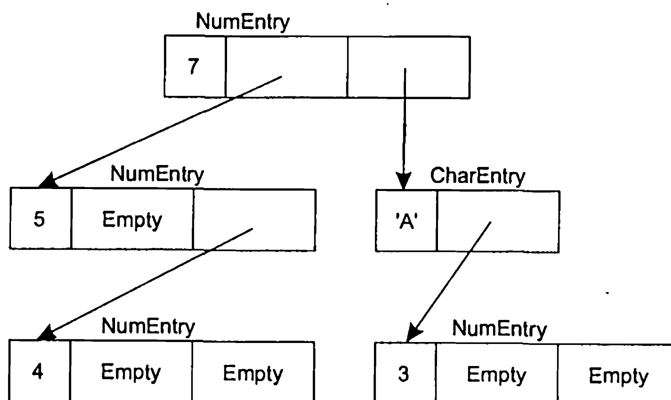


Рис. П.2. Древовидный тип данных языка ML

Она означает, что тип данных Tree состоит из трех частей: CharEntry, NumEntry и Empty. Структура рис. П.2 задается следующим образом:

```
- val NewTree=NumEntry(7,NumEntry(5,Empty,NumEntry(4,Empty,Empty)),
= CharEntry("A", NumEntry(3, Empty, Empty)));
val NewTree =
      NumEntry
         (7,NumEntry(5,Empty,NumEntry(4,Empty,Empty)),
         CharEntry("A",NumEntry(3,Empty,Empty))) : Tree
```

Доступ к компонентам типов данных можно продемонстрировать с помощью следующей программы:

```
datatype object = pair of int * int;
fun CreatePair(a,b) = pair(a,b);
fun CreateTwo(a,b,c,d) = CreatePair(a,b) :: CreatePair(c,d) :: [];
fun PrintFirst(pair(x,y)::z) = print(x)
| PrintFirst(x) = print("Error \n");
fun PrintThird(x::y) = PrintFirst(y)
| PrintThird(x) = print("Error \n");
```

Выполнение операции

```
val x = CreateTwo(1.2.3.4):
```

приводит к следующему результату:

```
val x = [pair (1.2).pair (3.4)] : object list
```

а выполнение следующих операторов:

```
PrintFirst(x):
PrintThird(x);
```

приводит к выводу

```
1
3
```

П.7.2. Управление последовательностью действий

Выражения

Арифметические выражения. Арифметические выражения используют следующий порядок старшинства для выражений:

~	Унарный минус
*, /, div, mod	Мультипликативные операторы
+, -	Аддитивные операторы

Можно использовать полный набор арифметических операций сравнения: =, <>, >=, <=, < и >.

Булевы выражения. Булевы выражения можно создавать, комбинируя следующие предикаты: andalso, orelse и not. Предикат andalso аналогичен предикату and, но разработан для «ленивого» вычисления: правый операнд вычисляется только в том случае, если левый принимает значение истина. Аналогично правый операнд операции orelse вычисляется, только если левый принимает значение ложь.

Строковые выражения. Конкатенация строк определяется знаком ^:

```
"abc" ^ "def" = "abcdef".
```

Пустая строка задается символами "".

Выражение if. В ML нет концепции *условного оператора*, так как это аппликативный язык. Вместо него используется условное выражение (как в языке C):

```
if выражение then в_случае_истины else в_случае_лжи
```

Поскольку каждое выражение должно быть правильно определено, часть else не является необязательной, как в операторах if других языков.

Выражение while. Выражение while задается следующим образом:

while выражение$_1$ do выражение$_2$

и имеет очевидную интерпретацию: выражение$_2$ выполняется до тех пор, пока истинно выражение$_1$. Однако как выражение$_1$ может изменить свое значение, если учитывать аппликативную природу выполнения программ на языке ML? Выражение$_2$ должно изменить ассоциацию переменной в локальном окружении. Также в языке ML имеется понятие *ссылочной переменной*, значение которой можно изменять.

Списочные выражения. Функции hd(L) для головы списка L и tl(L) для хвоста списка L осуществляют операции, подобные car и cdr в языке LISP. Операция cons языка LISP в языке ML задается двойным двоеточием ::. Например, hd(L) :: tl(L) = L.

В языке ML также существует операция объединения, которая соединяет два списка, [1.2]@[3.4] = [1.2.3.4]. Заметим, что для операции x :: y, если x имеет тип 'a, y должен быть типа 'a list, в то время как для объединения x@y обе переменные должны быть типа 'a list.

Приведение типов между списками и строками задается встроенными функциями explode и implode. Функция explode преобразует строку в список односимвольных строк, а функция implode получает в качестве параметра строковый список и возвращает строку.

Операторы

В языке ML фактически отсутствует понятие оператора, выполнение программы осуществляется как последовательность выполнения функций. Подобно тому, как выполняются программы на языке LISP, в ML выполнение программы осуществляется через рекурсивный вызов процедур. Каждое выражение завершается точкой с запятой (;).

Последовательное выполнение можно смоделировать выполнением последовательности

(выражение$_1$; выражение$_2$; ... ; выражение$_n$)

Каждое выражение вычисляется в порядке его задания.

Определение функций

Функции определяются следующим образом:

```
fun имя_функции (параметры) = выражение;
```

где выражение — это то выражение, которое следует вычислить. Параметры функции задаются в обычном контексте. Однако в отличие от других языков тип параметра можно опустить, если он выводим из контекста.

Например, сумму двух чисел можно определить так:

```
- fun sumit(a:int, b:int):int = a+b;
val sumit = fn : int * int -> int
```

Здесь указывается, что функция sumit получает в качестве параметров два целых (int) значения, возвращает также целочисленное значение (int), а имя sumit имеет тип function с сигнатурой int * int → int.

Любая из приведенных ниже спецификаций int однозначно определяет два других типа, так что следующие записи эквивалентны:

```
- fun sumit(a, b):int = a+b;
- fun sumit(a, b:int) = a+b;
- fun sumit(a:int, b) = a+b;
```

Однако необходимо задать хотя бы одну спецификацию int, поскольку запись

```
- fun sumit(a, b) = a+b;
```

неоднозначна, из нее не ясно, какой тип подразумевается — int или real.

Такой простой синтаксис позволяет определять мощные полиморфные функции. Рассмотрим функцию, которая перемешивает список путем перемещения головы списка в его конец. Она соединяет хвост списка со списком, состоящим только из его первого члена:

```
- fun shuffle(x) = tl(x) @ [hd(x)];
val shuffle = fn : 'a list -> 'a list
```

```
- shuffle([1.2,3]):
val it = [2.3.1] : int list
```

Функция shuffle оперирует с любым списком типа 'a и возвращает список того же типа. Однако когда shuffle применяется к конкретному списку [1.2,3], возвращается специальный тип int list.

В языке ML предусмотрена некоторая разновидность операции сопоставления с образцом для определения функций, как описано ранее в разделе 8.4.2. Можно определить собственную операцию соединения следующим образом:

```
fun append(a.b) = if a=nil then b else hd(a) :: append(tl(a).b);
```

Данную конструкцию можно рассматривать как две отдельных области, в которых определена функция append. Если a — пустой список nil, то возвращается значение b. Если a не пуст, то осуществляется другая операция. Язык ML позволяет определить отдельные области для функции:

```
fun app(nil.b) = b
| app(a.b) = hd(a) :: app(tl(a).b);
```

Для области, где первый аргумент nil, значением является b, в других случаях значением будет hd(a) :: app(tl(a).b).

Истинная сила операции сопоставления с образцом заключается в возможности доступа к компонентам сложных типов данных. Рассмотрим структуру, приведенную на рис. П.2. Объект X типа Tree определяется следующими компонентами:

```
NumEntry * Tree * Tree
CharEntry * Tree
Empty
```

Предположим, что мы хотим построить функцию last, которая имела бы доступ к самому правому компоненту рисунка. Ее можно определить так:

```
fun last(NumEntry(I.T1.T2))= if T2=Empty then if T1=Empty
        then NumEntry(I.T1.T2) else last(T1) else last(T2)
    | last(CharEntry(C.T1))= if T1=Empty then CharEntry(C.T1)
        else last(T1)
    | last(Empty) = Empty;
```

где каждая подобласть функции last рекурсивно выбирает самое правое поддерево. Когда мы применяем эту функцию к переменной NewTree, которую мы предварительно задали как структуру, изображенную на рисунке, мы получаем NumEntry с целым компонентом 3:

```
- last(NewTree);
val it = NumEntry (3.Empty.Empty) : Tree
```

Выражение as. Иногда бывает полезно в определении образца иметь возможность доступа к целому образцу. Это действие можно осуществить с помощью выражения as. Например,

```
fun AStart(L as "a"::b) = F1(L)
      AList(a) = F2(a)
```

вызовет:

1) F1 с аргументом L, если это список с первым элементом "a";

2) функцию F2 в противном случае.

Рекурсивные функции. Язык ML допускает рекурсивные функции, но, как и в большинстве других языков, косвенная рекурсия создает некоторые проблемы.

Если A вызывает B, а B вызывает A, какую из функций следует объявлять в первую очередь? В языке Pascal эта проблема разрешается с помощью объявления forward.

Поскольку в ML выполняется статическая проверка типов, то тип функции должен быть известен до того, как функция появится в выражении. Косвенная рекурсия осуществляется с помощью выражения and, которое приводит к тому, что все функции объявляются одновременно:

```
fun первая_рекурсивная_функция = ...
      and вторая_рекурсивная_функция = ...
      and последняя_рекурсивная_функция= ...;
```

Ввод-вывод

Функция print осуществляет простой вывод. Управляющую последовательность \n можно использовать для завершения строки вывода. Функция open_in(filename) открывает файл для ввода и возвращает указатель на дескриптор файла, имеющий тип instream. Функция input(file_ptr. n) считывает *n* символов из файла и возвращает строку string. Функция end_of_stream(file_ptr) возвращает значение истина, если достигнут конец файла.

Исключения

Язык ML позволяет пользователям определять исключения. Исключение определяется следующим образом:

```
exception имя_исключения
```

а генерируется так:

```
raise имя_исключения
```

Исключения можно обрабатывать с помощью оператора handle:

```
выражение handle обработчик
```

Если операция raise инициируется внутри выражения или из любой функции, вызываемой из выражения, то запускается обработчик. Он имеет формат:

```
обработчик(образец₁) => выражение₁
обработчик(образец₂) => выражение₂
...
обработчик(образецₙ) => выражениеₙ
```

Например, исключение myerror, порожденной оператором raise myerror(2), можно обработать следующим образом:

```
myerror(1) => выражение₁
myerror(2) => выражение₂
```

Стандартные функции

Функции ввода-вывода:

- use имя_файла считывает указанный файл как программу.
- print(x) печатает элементарный объект x.
- open in("имя_файла") открывает указанный файл и возвращает указатель типа instream на открытый файл.
- input(file_ptr. size) возвращает строку длиной size, состоящую из последующих size символов открытого файла file_ptr.

- end_of_stream(file_ptr) возвращает значение истина, если достигнут конец файла.

Другие функции:

- #i возвращает *i*-й компонент кортежа.
- hd(list) возвращает первый элемент списка.
- tl(list) возвращает список, состоящий из элементов списка list, начиная со второго и заканчивая последним.
- explode(string_val) преобразует строку string_val в список односимвольных строк.
- implode(list) получает список строк list и возвращает строку. Например, функция implode(["a", "bc"]) возвращает abc.
- size(str) возвращает количество символов в строке str.
- ord(x) возвращает целое значение символа x.
- chr(i) возвращает символ, представленный целым числом i.
- inc(i) эквивалентно i :=!i+1 для ref int i, а dec(i) эквивалентно i :=!i-1 для ref int i.

Абстракции и инкапсуляция

Структуры. ML включает концепцию пакетов, которые называются структурами (structure) и имеют следующий синтаксис:

```
structure идентификатор = struct
элемент₁
элемент₂
...
элементₙ end
```

где каждый элемент является объявлением типа, значения или функции. Например, простой пакет, который определяет функции для создания нового списка, добавления элемента в список, удаления элемента из списка и вычисления размера списка, можно описать следующим образом:

```
- structure ListItem = struct
= val NewList = nil;
= fun add([ ],y)= y :: nil |
=      add(x,y) = x @ [y];
= fun size(x) = if x=[ ] then 0 else 1+size(tl(x));
= fun delete(x) = tl(x)
= end;
```

Результирующая сигнатура будет иметь вид

```
structure ListItem :
       sig
          val NewList : 'a list
          val add : 'a list * 'a → 'a list
          val delete : 'a list → 'a list
          val size : 'a list → int
       end
```

Заметим, что структура полиморфна и может быть использована для создания списков любого типа (обозначенного как 'a) в сигнатуре.

Объявление open

```
open имя_структуры
```

создает динамическую область видимости для структуры и добавляет сигнатуру структуры в текущую среду, исключая необходимость явного именования структуры, как в приведенном списке целых переменных:

```
- open ListItem;
open ListItem
val add = fn : 'a list * 'a → 'a list
val size = fn : 'a list → int
val delete = fn : 'a list → 'a list
val NewList = [ ] : 'a list
- val a = NewList;
val a = [ ] : 'a list
- val b = add(a,3);
val b = [3] : int list
- val c = size(b);
val c = 1 : int
```

Сигнатура структуры определяет интерфейс подобно пакетам языка Ada или классам языка C++. Можно определить новую структуру, которая будет являться ограничением на более общую структуру. Например, можно определить списки int, которые будут ограничены списками целых величин и не будут содержать функцию delete, определяя эту новую сигнатуру при помощи объявления signature:

```
signature IntListSig = sig
       val NewList : int list;
       val add : int list * int → int list;
       val size : int list → int
end;
```

Можно определить вновь созданную структуру так, чтобы она выглядела, как исходная структура ListItem, только ограниченная до сигнатуры IntListSig, используя альтернативную форму структуры:

```
structure IntList : IntListSig = ListItem;
```

В этом месте можно вызвать NewList и add для создания новых целых списков, но delete не является частью нашей новой ограниченной среды:

```
- val i = IntList.NewList;
val i = [ ] : int list
- IntList.add(i,7);
val it = [7] : int list
- IntList.delete(i);
Error : unbound variable or constructor : in path
(Ошибка: несвязанная переменная или конструктор: delete в IntList.delete)
```

Абстракция. Использование объявления signature для создания ограничений на структуру является одним из способов сокрытия имен функций, как в предыдущем примере было скрыто имя функции delete в структуре IntList. Реализации Standard ML (Нью-Джерси) также включает конструкцию abstraction:

```
abstraction тип_абстракции : сигнатура_абстракции = определение_структуры
```

которая выполняет такую же функцию. Аналогично для создания новых типов данных можно использовать

```
abstype имя_типа_данных with объявления end
```

поскольку конструкторы объектов данных не экспортируются из определения типа, что обеспечивает истинную абстракцию.

Массивы. ML реализует массивы как структуры:

- open Array добавляет функции массивов в текущую среду.
- array(m, n) возвращает массив с диапазоном изменения индексов от 0 до (m-1), все элементы которого инициализированы как n.
- sub (a, i) возвращает значение a[i].
- update(a, i, val) присваивает элементу a[i] значение val.

П.8. Pascal

Пример с пояснениями

В листинге П.10 приведена версия на Pascal программы, которую мы уже рассматривали при изучении языков С и FORTRAN. В этой программе считывается массив целых чисел, распечатываются значения его элементов, а затем вычисляется и распечатывается сумма всех значений элементов массива. Если входной файл sample.data содержит

```
3
12 14 16
4 1.2 1.4 1.6 1.0
```

то в результате работы программы будет напечатано:

```
      12.00     14.00              16.00
sum = 42.0000
      1.20        1.40             1.60      1.00
sum = 5.2000
```

Листинг П.10. Пример суммирования элементов массива на языке Pascal

```
 1 program main(input, output, infile);
 2 const size = 99;
 3 type Vector = array [1..size] of real;
 4 var infile: text;
 5     a: Vector;
 6     j,k: integer;
 7 function sum(v: Vector; n: integer): real;
 8     var temp: real;
 9     i: integer;
10     {Тело функции sum}
11     begin
12     temp := 0;
13     for i := 1 to n do temp := temp + v [i];
14     sum := temp
15     end ;{sum}
16 begin {of main}
17     reset(infile, 'sample.data');
18     while not (eof(infile)) do
19        begin
20        read(infile, k);
21        for j := 1 to k do
22           begin
```

продолжение ⇒

Листинг П.10 (*продолжение*)

```
23             read(infile, a[j]);
24             write(a[j]:10:2)
25             end;
26          writeln;
27          writeln('sum = ', sum(a,k):6:4);
28          readln(infile)
29          end
30 end.
```

Строка 1. main — это имя программы. Файл input ссылается на стандартный файл ввода (обычно это клавиатура), output — на стандартный файл вывода (обычно это экран монитора), а infile — на файл данных, который используется внутри программы. Здесь необходимо упоминать только файлы, используемые в программе.

Строка 3. Определяется новый тип данных Vector, который представляет собой массив вещественных чисел с диапазоном изменения индекса от 1 до 99. И верхняя, и нижняя границы массива всегда должны указываться.

Строки 4–6. Оператор var определяет глобальные переменные, используемые в программе. В данном случае infile — это текстовый файл, содержащий входные данные. Файлы input и output объявлять не требуется.

Строка 7. Объявляется функция sum. У нее имеются параметры v типа Vector и n типа integer, передаваемые по значению. Если бы требовалось передавать параметры по ссылке, то в объявлении функции надо было бы написать, например: var v: Vector. Функция sum возвращает значение типа real (вещественное число).

Строка 10. Комментарии в Pascal задаются в фигурных скобках.

Строка 13. Оператор for, в котором переменная цикла i изменяется от 1 до n. Если тело оператора for состоит более чем из одного оператора, то всю последовательность операторов, составляющую тело цикла, требуется помещать между операторами begin и end (см. строки 22–25).

Строка 14. Функция возвращает вычисленное значение простым его присваиванием имени функции.

Строка 17. Оператор reset открывает файл для ввода. В данном случае он связывает файл sample.data с внутренним объектом данных infile. Если данные вводятся стандартным образом с клавиатуры, то этот оператор не нужен. Оператор rewrite открывает файл для записи.

Строка 20. Здесь считывается первое значение из файла infile и сохраняется в переменной k. Если бы ввод осуществлялся с клавиатуры, то имя файла можно было опустить. Это — ввод в свободной форме. Pascal считывает символы до запятой или пробела и пытается перевести прочитанные данные в целочисленный формат. Если бы переменная k была символьного типа (char), то был бы прочитан только один символ из файла ввода.

Строка 24. Значение элемента вектора записывается в стандартный файл вывода (то есть отображается на экране монитора), так как в операторе write первым параметром не задано имя файла, в который следует записывать данные. Необязательные члены 10:2 предписывают печатать 10 десятичных цифр числа, причем две цифры отводятся под дробную часть. Если эти параметры не указаны, то Pascal использует умалчиваемый формат представления десятичных чисел. Следующие операторы write записывают числа в ту же строку вывода до тех пор, пока не будет выполнен оператор writeln (строка 26).

Строка 26. Оператор writeln добавляет к предыдущему выводу символ конца строки и завершает строку вывода (печать значений массива).

Строка 27. Здесь печатается строка "sum = ", вызывается функция sum для вычисления суммы, а затем вычисленное значение выводится на экран в виде числа, состоящего из шести цифр с четырьмя знаками после десятичной точки. Поскольку для вывода используется оператор writeln, то после этого числа добавляется символ конца строки. Если не требуется печатать результат отдельной строкой, то можно использовать оператор write.

Строка 28. Из файла infile читается следующая строка.

Строка 30. Конец файла программы обозначается точкой после ключевого слова end.

П.8.1. Объекты данных

Pascal является наиболее строго типизированным языком; границы массивов, например, объявляются как часть типа. Спецификации различных элементарных типов данных в Pascal в основном достаточно очевидны, а некоторые простые ограничения имеют целью обеспечение эффективности реализации.

Элементарные типы данных

Переменные и константы. Каждое имя переменной (как и большинство других идентификаторов), используемое в программе на Pascal, должно быть явным образом объявлено в начале программы или подпрограммы. Для любых локальных и нелокальных ссылок, используемых в каждой операции, выполняется полная статическая проверка типов.

Синтаксис Pascal требует, чтобы в первую очередь были объявлены константы, затем локальные переменные и подпрограммы. Тем не менее во многих компиляторах эти требования не столь жесткие и допускается объявлять различные программные объекты в произвольном порядке.

Константы объявляются следующим образом:

```
constant_name = constant_value
```

Заметим, что в правой части этого объявления не должно стоять выражение, то есть константа не может быть определена через значение другой константы.

Числовые типы данных. В Pascal предусмотрены целочисленные объекты данных типа integer, для которых обычно используется аппаратное представление целых чисел. Арифметические операции и операции сравнения определены с помощью обычной инфиксной записи для бинарных операций: +, -, div (деление), mod (остаток от деления), =, <> (неравенство), <, >, <= (меньше или равно) и >= (больше или равно). Наибольшее допустимое целое число равно значению стандартной константы maxint (устанавливается разработчиком языка для отражения наибольшего целого значения, которое может быть представлено в компьютере).

Стандартный тип real соответствует аппаратной реализации вещественных чисел с плавающей точкой. Предусмотрен простой набор операций, также использующих инфиксную запись: +, -, =, <>, <, >, <=, >= и / (деление). Операция возведения в степень не определена, но встроены некоторые основные арифметические функции, например sin (синус), cos (косинус) и abs (абсолютное значение).

Перечисляемый тип определяется с использованием следующего синтаксиса:

```
type_name = (literal1, literal2,..., literalk)
```

где $literal_i$ — выбранные пользователем идентификаторы. Эти идентификаторы представляют собой упорядоченную по возрастанию последовательность значений, так что $literal_1 < literal_2$.

Поддиапазон последовательности целых значений или значений из перечисляемого типа задается следующим образом:

```
первое_значение..последнее_значение
```

(например, 2..10 или Soph..Senior). Перечисление или поддиапазон перечисления можно использовать для задания диапазона изменения индексов массива. Например, объявление

```
Q:array [Class] of integer;
```

задает массив из четырех компонентов с индексами, принадлежащими типу Class: Q[Fresh], Q[Soph] и т. д. (см. раздел 6.1.5).

Логический тип данных. Предопределенный логический (булев) тип данных задается как перечисление:

```
boolean = (false,true)
```

Для этого типа данных определены элементарные операции and, or и not, так же как операции сравнения (и другие операции), определенные для любого перечисления. Булевы значения реализованы как целые 0 и 1.

Символьный тип данных. Стандартный тип char определен как созданное при реализации перечисление, представляющее упорядоченную последовательность допустимых в данной реализации символов. Помимо операций отношения succ, pred и ord, свойственных любому перечислению, для символьного типа данных определена операция chr, операндом которой является целое число из промежутка от 0 до числа, равного количеству символов в данной последовательности, уменьшенному на единицу, а результатом — соответствующий этому числу символ.

Указатели. Объект данных может содержать указатель на другой объект данных. Тогда тип первого объекта данных объявляется как ↑ второй_тип, где второй_тип — это тип того объекта данных, указатель на который и может хранить первый объект. Переменная-указатель может быть определена следующим образом:

```
имя_переменной: ↑ тип
```

и тогда значением этой переменной может быть либо пустой указатель, обозначаемый как nil, либо фактический указатель на какой-либо объект данных того типа, который задан в объявлении.

Для указателей определены только следующие операции: присваивание (которое сводится к копированию значения указателя), операции сравнения = и <> (не равно), предназначенные для сравнения значений указателей, а также операция new, которая создает объект данных указанного типа и возвращает указатель на этот объект.

Структурированные типы данных

Массивы. Базовая структура данных типа массив в Pascal имеет только одну размерность, а диапазон значений индексов может быть определен как произвольное упорядоченное подмножество целых чисел или перечисление. Диапазон индексов

должен быть определен во время компиляции, что позволяет компилятору произвести все вычисления, связанные с адресацией элементов массива, во время компиляции, и не использовать дескрипторы во время выполнения программы. Элементы массива могут быть любого типа, как элементарного, так и определенного программистом. Поэтому допустимыми являются такие конструкции, как массив записей, массив множеств или массив файлов. Массив объявляется следующим образом:

```
array [диапазон_индексов] of тип_компонентов
```

Многомерный массив конструируется из одномерного массива, элементами которого являются также одномерные массивы (то есть вектор, состоящий из векторов). Например, определить матрицу размером 3 × 6, состоящую из вещественных чисел, можно при помощи следующего объявления:

```
array [1..3] of array [1..6] of real
```

Для удобства записи допускается сокращение объявления многомерного массива с помощью перечисления всех диапазонов изменения индексов в одном месте:

```
array [1..3,1..6] of real
```

Эта запись позволяет также использовать более простой синтаксис индексации при выборке элемента массива (например, A[3,4] вместо A[3][4]).

Строка символов в Pascal (возможно, с необязательным указанием ее упаковки packed) представляется как массив, каждый элемент которого — это отдельный символ. Поскольку диапазон изменения индексов массива уже зафиксирован во время компиляции, такое представление строк соответствует представлению строк фиксированной длины, описанному в главе 5. Единственными допустимыми операциями для символьных строк являются операции отношения (=, <, > и т. п.), применяемые к строкам одинаковой длины. Результат определяется в соответствии с упорядочением символов в перечислении, реализующем тип char (см. обсуждение ранее символьного типа данных). Допускается присваивание значения переменной, представляющей символьную строку, но это значение должно быть символьной строкой той же длины. Символьные строки могут быть выведены в текстовые файлы, но читаются они посимвольно в обычную переменную или массив типа char.

Записи. Базовый тип данных запись достаточно подробно описан в разделе 6.1.6.

Множества. Pascal предоставляет ограниченную форму типа данных множество. Объект данных этого типа состоит из множества компонентов, каждый их которых принадлежит некоторому базовому типу данных, который ограничен перечислением и поддиапазоном целых чисел, содержащим количество значений, ограниченное некоторым максимальным числом. Это число обычно определяется при реализации языка и соответствует количеству бит в одном или нескольких словах памяти компьютера, для которого предназначена данная реализация. Тогда множество может быть реализовано в виде последовательности булевых значений (единичных битов), в которой каждый бит соответствует определенному элементу базового множества. Если бит равен 1 (истина), то этот элемент входит в множество, в противном случае элемент в множество не входит.

Синтаксис для объявления объекта данных типа множество следующий:

```
имя_типа = set of базовый_тип
```

(базовый тип был описан ранее). Затем переменная типа множество может принимать значения, соответствующие любому подмножеству значений базового типа, включая пустое множество (записывается как []). Литеральная константа, представляющая множество, может быть записана в следующем виде:

```
[value.....value]
```

где value — либо единичное значение, либо подмножество значений из базового множества.

Для множеств предусмотрены следующие операции: объединение (синтаксически обозначается как +), пересечение (обозначается через *) и разность (-). Также для множеств определены операции отношения =, <>, <= и >=, причем две последние обозначают проверку вхождения одного множества в другое (например, запись A <= B означает: является ли A подмножеством B?). Чтобы проверить, является ли конкретное значение элементом множества, используется операция in. Например, Soph in S означает: входит ли значение Soph в множество, являющееся текущим значением переменной S?

Файлы и ввод-вывод. Последовательные файлы — это единственная файловая структура, предусмотренная в Pascal. Для объявления файла используется синтаксис

```
file of тип_компонентов
```

где тип_компонетов может быть любым типом, за исключением файлов и указателей. Таким образом, помимо объектов данных элементарных типов компонентами файла могут быть массивы, записи и другие объекты данных структурированных типов.

Первая строка программы должна иметь следующий синтаксис:

```
program имя_программы (список_файлов)
```

где список_файлов — это список имен файлов из некоторой внешней для программы среды. Эти файлы должны быть локально объявлены в основной программе. Другие локальные файлы могут быть объявлены в программе или в подпрограммах так же, как и переменные любого другого типа. Эти локальные файлы можно использовать в качестве временных файлов, куда в одной части программы записывается информация, которая затем считывается в другой ее части. Локальные файлы исчезают при выходе из программы или подпрограммы, в которой они были объявлены, как и любой другой локально определенный объект в Pascal.

При объявлении файла его имя используется как имя буферной переменной, ссылка на которую выглядит как имя_файла↑. Эта буферная переменная не объявляется явным образом, но в программе можно на нее ссылаться и использовать в операторах присваивания. Например, если переменная F объявлена как

```
F: file of integer
```

то внутри той программы, где содержится это объявление, операторы присваивания F↑:= 2 и X:= F↑ будут вполне корректными. Буферная переменная — это объект данных, представляющий локальную временную область памяти для одного компонента файла. Две предопределенные процедуры get и put используются для перемещения данных из файла в буферную переменную (get) и из буферной переменной в файл (put).

Определяемые пользователем типы данных

В Pascal новый тип данных создается с помощью объявления:

```
type имя_типа = определение_типа
```

Определения типов в Pascal используются только во время компиляции. Компилятор вносит информацию, полученную из определений типов, в свою таблицу символов, а затем использует ее для осуществления статической проверки типов в ссылках на переменные и для компоновки записей активаций подпрограмм. Во время выполнения программы определения типов отсутствуют.

В некоторых случаях считается, что две переменные относятся к одному типу данных, если в их объявлениях указано одно и то же имя типа. В других случаях считается, что переменные относятся к одному типу, если типы, фигурирующие в их объявлениях, имеют одну и ту же структуру. Так, для определения того, относятся ли две переменные к одному и тому же типу, в Pascal используется объединение двух понятий эквивалентности типов — эквивалентность *имени* и эквивалентность *структуры*. Эквивалентность имени используется в основном для определения того, совпадают ли типы формальных и фактических параметров при вызове подпрограммы; в большинстве других ситуаций используется структурная эквивалентность.

П.8.2. Управление последовательностью действий

Выражения

В Pascal используются четыре уровня старшинства операций для вычисления выражений. Их иерархия в порядке убывания приоритетов выглядит так:

```
not
*, /, div, mod, and
+, -, or
=, <>, >=, <=, <=, >=, in[1]
```

В пределах одного уровня иерархии члены выражения вычисляются слева направо. Обратите внимание на то, что по сравнению с обычным вычислением выражений этот способ выглядит несколько противоестественно. Обычно приоритет операции and ниже, чем приоритет арифметических операций. Например, следующий фрагмент:

```
var A,B:boolean;
...
if A and B=false then ...
```

будет интерпретирован компилятором Pascal как

```
if (A and B)=false then ...
```

хотя, вероятно, имелось в виду следующее:

```
if A and (B=false) then ...
```

---

[1] Это не опечатка: здесь, судя по всему, авторы выделили три операции сравнения для множеств (<=, >= и in), поэтому символы употреблены дважды. — *Примеч. науч. ред.*

Условие в первой интерпретации будет выполнено, если либо A, либо B имеют значение false, в то время как во второй интерпретации предполагается, что A имеет значение true, а B — false. Поэтому лучше, во избежание подобной путаницы, всегда использовать скобки в логических выражениях.

В выражениях Pascal не выполняется никакого преобразования типов, за исключением преобразования целочисленных операндов в вещественные, когда это следует из контекста.

Операторы

Составные операторы. В Pascal составной оператор ограничен конструкцией begin ... end:

```
begin
      оператор
      ...;
      оператор
end
```

Условные операторы. В Pascal имеются два условных оператора: if и case.

Оператор if. Простой условный оператор с одной и двумя ветвями выглядит следующим образом:

```
if логическое_выражение then оператор
if логическое_выражение then оператор else оператор
```

Если две эти формы объединены, как, например, в следующем случае:

```
if A = 0 then if B < C then S1 else S2;
```

то возникающая неоднозначность, которая обсуждалась разделе 8.3.2, всегда разрешается таким образом, что часть else всегда соотносится с ближайшим предшествующим несогласованным then.

Оператор case. Оператор case имеет следующую форму:

```
case выражение of
      константа: оператор;
      ...
      константа: оператор
end
```

где константы представляют возможные значения выражения.

При выполнении оператора case вычисляется выражение, расположенное в начале этого оператора, и полученное значение используется для определения того, какой из следующих операторов должен выполняться. Если в результате вычисления выражения получается значение, которое не равно ни одной из констант оператора case, то эта ситуация рассматривается как ошибка времени выполнения программы.

Операторы цикла. В Pascal имеется три вида операторов цикла. В двух основных операторах, while и repeat, проверка условия окончания цикла проводится *перед* очередной итерацией (while) или *после* каждой итерации (repeat). Синтаксис этих операторов следующий:

```
while логическое_выражение do оператор
repeat последовательность_операторов until логическое_выражение
```

В обоих случаях тело цикла может быть представлено последовательностью операторов, но для оператора while требуется явным образом определить тело цикла с помощью составного оператора begin ... end.

Третий вид операторов цикла — это цикл for, в котором заданное число итераций сопровождается последовательным увеличением или уменьшением счетчика. Синтаксис обеих форм оператора for следующий:

```
for простая_переменная := начальное_значение to конечное_значение do оператор
for простая_переменная := начальное_значение downto конечное_значение do оператор
```

Здесь форма to указывает, что после каждой итерации простая_переменная (счетчик) возрастает на единицу, а форма downto указывает на соответствующее уменьшение этой переменной на единицу. Другие варианты увеличения или уменьшения счетчика в Pascal не допускаются. Начальное и конечное значения задаются произвольными выражениями, которые вычисляются при входе в цикл. Внутри цикла запрещается присваивание каких-либо значений переменной, контролирующей итерации.

Конструкции, подверженные ошибкам. Оператор Pascal можно пометить с помощью целочисленной метки и затем передать ему управление при помощи оператора goto. Как неоднократно подчеркивалось в нашей книге, использование этой конструкции не является необходимым и, вообще говоря, приводит к сложностям при отладке программ.

Операторы ввода-вывода. Для большинства операций ввода-вывода функции низкого уровня get и put, а также операторы read и write обеспечивают всю необходимую функциональность.

Функции get и put. Для записи в файл очередного компонента необходимо предпринять следующие шаги:

1) значение нового компонента присвоить буферной переменной этого файла;
2) вызвать процедуру put, которая передаст данные из буферной переменной в файл, вставляя новый компонент с заданным значением в конец файла и перемещая указатель текущей позиции в файле на следующую за новым компонентом позицию.

Например, для записи в файл F значения 2 требуется выполнить два оператора: F↑:=2;put(F). Для считывания текущего компонента файла в локальную переменную требуется выполнить аналогичную последовательность действий (например, X:=F↑;get(F) копирует текущий компонент из буферной переменной F↑ в переменную X, а затем перемещает указатель текущей позиции файла к следующему компоненту, помещая его значение в буферную переменную F↑).

Операторы read *и* write. Поскольку обработка файлов обычно требует многократных повторений описанной последовательности двух шагов, в Pascal имеются стандартные операторы read и write, которые объединяют соответствующую последовательность в вызов единой процедуры. Например, read(F, X) эквивалентно X:=F↑;get(F).

Операторы readln *и* writeln *и текстовые файлы.* Текстовый файл — это файл, компоненты которого представляют собой отдельные символы; он объявляется с использованием стандартного типа text. Предполагается, что текстовый файл организован в строки, которые являются последовательностями символов, завершающимися специальным символом конца строки. Текстовые файлы можно обрабатывать с помощью обычных файловых операций get и put, а также с помощью операторов write и read. Но эти операторы расширены таким образом, что допускают использование в качестве параметров данных, отличных от символьного типа (например, read(F,N), где N — целочисленная переменная). Когда параметром опе-

ратора write или read оказывается объект числового, а не символьного типа, то этот оператор автоматически преобразует символьное представление компонента файла во внутреннее двоичное представление, требуемое в качестве значения переменной. Оператор read сканирует входной файл, пока не будет найдено необходимое для него полное число компонентов требуемого типа; оператор write обеспечивает автоматическое форматирование выводимых данных при записи их в файл вывода. Процедуры writeln и readln выполняют те же действия, что и операторы write или read, но дополнительно перед своим завершением переводят указатель текущей позиции в файле на новую строку.

Стандартные функции

Функции ввода-вывода:

- Функция rewrite(f) открывает файл f для вывода. Во многих реализациях эта функция задается как rewrite(f, имя_файла), связывающая внешнюю строку имя_файла с внутренним файлом языка Pascal f. reset(f), или reset (f, имя_файла), открывающая файл f для ввода.
- Функция put(f) перемещает объект данных из буферной переменной файла f↑ в файл вывода. Функция get(f) перемещает очередной объект данных из файла в буферную переменную f↑. Функции read, write, readln и writeln в действительности являются процедурами и уже были описаны в предыдущем разделе, посвященном операторам Pascal.
- Функция eof(f) возвращает значение true, если указатель позиции в файле f указывает на его конец. Если f является файлом ввода, то функцию можно записать просто как eof. Функция eoln(f) возвращает значение true, если из файла f читается символ конца строки. К сожалению, в Pascal символ конца строки всегда трансформируется в пробел, так что его присутствие невозможно проверить явным образом, как в С.

Функции управления памятью:

- Функция new(p) отводит место в памяти для объекта данных, тип которого совпадает с типом объекта данных, на который указывает переменная-указатель p, и устанавливает *r*-значение p так, чтобы оно указывало на этот объект данных.
- Функция dispose(p) освобождает память, занятую под объект данных, на который указывает указатель p. Эта память должна была быть выделена ранее при помощи вызова функции new(p).

Арифметические функции:

- Функции abs(x), sqr(x), sin(x), cos(x), exp(x), ln(x), sqrt(x) и arctan(x) вычисляют соответственно абсолютное значение, квадрат, синус, косинус, экспоненту (e^x), натуральный логарифм, квадратный корень и арктангенс своего аргумента.
- Функции trunc и round — это две функции для преобразования вещественных данных в целочисленные. trunc(x) возвращает целочисленное значение, причем $0 \le x - trunc(x) < 1$, если x положительно или равно 0, и $-1 < x - trunc(x) \le 0$, если x меньше 0. round(x) = trunc(x+0.5), если x положительно или равно 0, и round(x) = = trunc(x-0.5), если x отрицательно.

- Функции pack и unpack используются для преобразования символьных массивов в упакованные массивы. pack(a, i, z) означает, что следует создать упакованный массив z из компонентов массива a, начиная с a[i]. unpack(z, a, i) означает, что нужно распаковать массив z, формируя массив a, начиная с элемента a[i].

Порядковые функции:

- Функция ord(x) возвращает индекс параметра x как элемента перечисления.
- Функция chr(x) возвращает символ, который представлен целым числом x.
- Функция succ(x) возвращает значение элемента, следующего за x в перечислении. Если следующего элемента нет, вызов этой функции воспринимается как ошибка.
- Функция pred(x) возвращает значение предыдущего элемента в перечислении. Аналогично если предыдущего элемента нет, то вызов этой функции приводит к ошибке.

П.9. Perl

Пример с пояснениями

В листинге П.11 приведена программа на Perl, суммирующая элементы массива. В данном примере при вводе данных

```
1 2 3 4
```

будет получен следующий результат:

```
1234
SUM = 10
```

Программа на Perl состоит из последовательности операторов, каждый их которых завершается точкой с запятой. Первая строка программы является псевдокомментарием — она начинается с символа #, который используется для обозначения комментариев, за которым сразу же следует символ !. В этой строке указывается имя программы, которая будет выполнять сценарий. Обычно первая строка выглядит так[1]:

```
#!/usr/bin/perl
```

Листинг П.11. Пример программы суммирования элементов массива на языке Perl

```
1 #!/usr/bin/perl
2 @inputdata = split(/ /, <STDIN>);
3 $count = 0;
4 foreach $nextone (@inputdata)
5     {print "$nextone";
6     $count = $count + $nextone;};
7 print "Sum = ";
8 print "$count\n";
```

[1] Это относится к сценариям Perl, выполняемым в операционной системе UNIX. В системе Windows псевдокомментарий обычно игнорируется, но может быть использован для установки режимов работы интерпретатора perl заданием в этой строке соответствующих его ключей. — *Примеч. науч. ред.*

Строка 1. Информирует операционную систему о том, что данный сценарий будет выполняться интерпретатором perl.

Строка 2. Читает вводимую пользователем строку данных и разбивает ее на элементы массива @inputdata, используя в качестве разделителя пробелы.

Строки 4–6. Выполняется цикл по всем элементам массива @inputdata, на каждом шаге которого значение очередного элемента массива добавляется к значению переменной $count.

Строка 4. На каждом шаге цикла foreach переменной $nextone присваивается значение очередного элемента массива @inputdata.

Строки 7–8. Распечатывается вычисленная сумма, а также символ конца строки.

П.9.1. Объекты данных

Элементарные данные. Скалярные переменные имеют идентификаторы, начинающиеся с символа $, и могут хранить числовые и строковые данные. Строки заключаются в одинарные кавычки (' '). Для вычисления значения строки при ее обработке интерпретатором[1] она должна быть заключена в двойные кавычки (" "). Следовательно, при выполнении

```
$x = 'mvz':
print 'Мои инициалы - $x':
```

будет напечатано Мои инициалы - $x, в то время как операторы

```
$x = 'mvz':
print "Мои инициалы - $x";
```

напечатают Мои инициалы - mvz. Для многих функций, если не задан параметр, по умолчанию используется параметр $_.

Логические величины, как и в С, являются целочисленными значениями. Ноль соответствует значению ложь, а ненулевое значение соответствует истине.

Скалярные массивы. Перед именами массивов ставится символ @[2]. Массивы скаляров можно определять неявным образом, присваивая имени массива соответствующее значение. Например, присваивание

```
@classStanding = ('freshman'.'sophomore'.'junior'.'senior'):
```

создает массив из четырех элементов с индексами от 0 до 3. Для доступа к элементу массива используется символ $ перед именем массива (например, $classStanding[2]), так как элемент массива является скалярным типом.

Массивы — динамические объекты. С помощью функции push к массиву можно добавлять элементы, как, например, в push (@classStanding, 'gradstudent'), где к массиву classStanding добавляется пятый элемент gradstudent.

Ассоциативные массивы. Массивы можно рассматривать как отношения между *ключом* и *значением*. В Perl ассоциативные массивы используются для фиксирования этих отношений. Такие массивы обозначаются символом % перед их идентификатором. Таким образом, запись

---

[1] В Perl строка в двойных кавычках на самом деле определяет операцию подстановки в нее значений скалярных переменных и массивов скаляров, заданных в ней. — *Примеч. науч. ред.*

[2] Так определяются массивы скаляров, позволяющие хранить числовые и строковые значения, то есть скалярные данные. — *Примеч. науч. ред.*

```
%GPA = ('Tom', 3.6,
'George', 2.6,
'Sam', 3.2)
```

создает ассоциативный массив из трех элементов. Далее, оператор

```
print "GPA of Tom is $GPA{'Tom'}";
```

напечатает GPA of Tom is 3.6. Заметим, что в этом случае перед идентификатором ассоциативного массива (которому обычно предшествует символ %) стоит символ $. Это объясняется тем, что в данном случае мы ссылаемся на элемент массива, а это — скалярный объект.

Использование foreach $name (key %GPA) позволяет программе последовательно обработать каждый элемент ассоциативного массива, то есть обработать *поле значений* массива[1].

Регулярные выражения. Регулярное выражение начинается и заканчивается косой чертой[2] и используется в качестве правого операнда операций сопоставления с образцом, в частности операции связывания =~. Результатом операции $x =~ thing будет истина, если строковая переменная $x содержит строку thing. Результатом операции связывания !~ будет истина, если строка, заданная регулярным выражением, не будет найдена.

В регулярных выражениях могут использоваться следующие метасимволы и конструкции:

.	Соответствует любому единичному символу, кроме символа новой строки
^	Соответствует началу строки
$	Соответствуют концу строки
*	Соответствует фрагменту текста, в котором предшествующий символ встречается ноль или более раз
+	Соответствует фрагменту текста, в котором предшествующий символ встречается один или более раз
?	Соответствует фрагменту текста, в котором предшествующий символ встречается ноль или один раз
[abc]	Соответствует любому из указанных символов a, b или c
[^abc]	Соответствует любому символу, за исключением a, b или c
[a-z]	Соответствует любому символу из указанного диапазона
a|b	Соответствует одному из символов, a или b

Те метасимволы, которые используются в образцах (например, |, ^), можно задавать в регулярных выражениях, если поставить перед такими символами экранирующий символ \.

Если группу элементов регулярного выражения заключить в круглые скобки (например, /([a-z])/), то найденное ей соответствие в строке, сопоставляемой с образцом, будет помещено в переменную $1 (для следующей группы в круглых скобках будет использована переменная $2 и т. д. до $9). Это свойство используется при операции подстановки.

---

[1] Стандартная функция key возвращает скалярный массив значений ассоциативного массива, переданного ей в качестве параметра. — *Примеч. науч. ред.*

[2] Обычно используется этот символ, хотя на самом деле при задании регулярного выражения в качестве ограничителя можно использовать любой символ, один и тот же в начале и в конце. — *Примеч. науч. ред.*

Подстановка. Конструкция s/рег_выраж/значение/ имеет следующий смысл: если в сопоставляемой строке находится последовательность символов, удовлетворяющая регулярному выражению рег_выраж, то она заменяется заданным строковым значением. Конструкция s/рег_выраж/значение/g производит замену во всех случаях такого соответствия. Переменные Perl с $1 по $9 также можно использовать для подстановки, используя их в замещающем значении; внутри самого регулярного выражения для ссылки на содержимое такой переменной следует использовать конструкцию вида \1, \2 и т. д. до \9.

П.9.2. Управление последовательностью действий

Операторы. В Perl используются обычные управляющие структуры. В каждый конкретный момент времени выполняется не более одного оператора. Каждый оператор заканчивается точкой с запятой. Оператор присваивания Perl является достаточно обычным, но в применении к массивам он имеет некоторые особенности:

```
@x = @y;          #Присваивание массива у массиву x
$x = @y;          #Присваивание длины массива у переменной x
$x = "@y";        #Подстановка в строку через пробел значений
                  #элементов массива у. Полученная строка
                  #присваивается переменной x
```

Конструкция цикла

```
foreach $indexvar (@arrayvalue) {оператор};
```

последовательно на каждом шаге присваивает переменной $indexvar значения элементов массива и выполняет оператор. В Perl, как и в С, имеется оператор for, а также операторы while-do и do-until. Предикаты задаются в соответствии с соглашениями, принятыми в С. Равенству соответствует операция ==, неравенству — операция !=, равенство строк определяется операцией eq, а неравенство строк — операцией ne.

Общий вид условного оператора if следующий:

```
if (выражение)
      then {оператор}
      elseif (выражение) {первый возможный оператор}
      elseif (выражение) {второй возможный оператор}
      else {последний возможный оператор};
```

Считывание из файла требует использования дескриптора файла, так

```
$x = <STDIN>;
```

считывает очередные входные данные из стандартного файла ввода (например, с клавиатуры). Оператор

```
open (filename. $stringvalue);
```

открывает файл с именем, содержащимся в переменной $stringvalue, и создает дескриптор файла filename, а оператор

```
close(filename)
```

закрывает этот файл. В Perl используются также соглашения оболочки shell по доступу к файлам UNIX:

```
open(myfile,">$file");          #Открыть файл для вывода данных
open(myfile,">>$file");         #Открыть файл в режиме добавления
open(myfile,"<$file");          #Открыть файл для ввода данных
```

Функции. Функция push(массив, скалярная_величина) добавляет элемент к массиву, а функция pop(массив) возвращает последний элемент массива.

Функция key возвращает первые члены отношений ассоциативного массива (ключи), в то время как value возвращает вторые члены этих отношений (значения).

Функция split(rexpr,stringvar) использует регулярное выражение rexpr в качестве разделителя и преобразует строковое выражение stringvar в массив. Результатом выполнения оператора @x=split(/ /, $y); будет разделение строки $y на подстроки (каждая подстрока — это символы между двумя пробелами в строке) и создание массива, элементами которого являются полученные подстроки. Если строковое выражение не задано, то по умолчанию вместо него используется содержимое специальной переменной $_.

Функция chop удаляет из строки символ конца строки.

Функция tr (транслитерация) преобразует строку из одного множества символов в другое. Например, tr/a-z/A-Z/ преобразует содержимое специальной переменной $_ из нижнего регистра в верхний.

Подпрограммы. Синтаксис подпрограмм следующий:

```
sub имя { операторы }
```

Вызов подпрограмм осуществляется как &имя, а параметры передаются через элементы специального массива @_. Возвращаемое значение — то, которое было вычислено в подпрограмме последним. Например,

```
local($x, $y);
```

создает в подпрограмме локальные переменные.

П.10. Postscript

П.10.1. Объекты данных

Элементарные данные. Данные в Postscript включают в себя следующие объекты:

Числа могут быть целыми (например, 1, 3, -42) или вещественными (например, -1.234, 1.2E-10).

Листинг П.12. Программа на Postscript

```
 1: %Аналогична программе на Forth, приведенной в начале раздела 8.2.2
 2: /Helvetica findfont
 3: 20 scalefont
 4: setfont
 5: 200 400 moveto
 6: /formatit {10 10 string cvrs show } def
 7: /sqr {dup mul} def
 8: /dosum {exch 1 add exch 1 index sqr add} def
 9: 3 6 dosum 2 copy formatit ( ) show formatit
10: clear
11: 200 375 moveto
12: 0 0 0 1 9 {pop dosum} for formatit
13: showpage
```

Составные данные. Составные данные могут включать в себя несколько компонентов, сгруппированных вместе, которые, в свою очередь, состоят из следующих элементов:

Массивы — элементарные объекты данных, заключенные в квадратные скобки ([,]), например:

```
[123 4 567] %Массив из трех чисел
```

(комментарии начинаются с символа %) или

```
[123 abc /xyz] %Массив из целого числа, имени и литерального имени
```

Строки представляют собой произвольную последовательность символов, заключенную в круглые скобки:

```
(Это строка)
```

Процедуры — это последовательности лексем Postscript, заключенные в фигурные скобки ({, }):

```
{dup mul} %Возводит число в квадрат, дублируя верхний элемент стека и затем перемножая
          два верхних (одинаковых) элемента
```

В качестве примера программы на Postscript приводится программа, аналогичная программе на языке Forth из обзора языка 8.2.

Строка 1. Комментарий в Postscript начинается с символа %.

Строка 2. Имя (литерал) Helvetica помещается в стек операндов, а затем функция findfont заменяет это имя на определение шрифта в стеке операндов. Helvetica — это достаточно распространенный шрифт без засечек. Типичным шрифтом с засечками является шрифт Times Roman.

Строка 3. Функция scalefont масштабирует шрифт Helvetica до кегля 20 (20 точек). В обычных текстах используется шрифт кегля 10 или 11. Расстояния измеряются в точках: одному дюйму соответствует 72 точки.

Строка 4. Шрифт, заданный в стеке операндов (Helvetica кегля 20), становится текущим шрифтом для программы. Весь текст, который печатает программа, будет печататься именно этим шрифтом, если не будет задан новый шрифт.

Строка 5. Курсор перемещается в точку с координатами (200.400), то есть сдвигается примерно на 3 дюйма вправо от левого края страницы и на 6 дюймов вверх от нижнего края. Начало координат (точка (0.0)) располагается в левом нижнем углу страницы.

Строка 6. formatit определяется как функция в пользовательском словаре userdict, которая помещает целое, расположенное в вершине стека, на страницу. Она создает строку из 10 нулей (10 string), берет верхний элемент из стека операндов, преобразует его в десятичное число, помещает его в строку (cvrs) и затем закрашивает эту строку, начиная с текущей позиции курсора на странице (show).

Строка 7. sqr определяется как функция, которая дублирует верхний элемент в стеке и перемножает два верхних элемента (то есть возводит этот элемент в квадрат).

Строка 8. dosum определяется как функция, которая получает в качестве операндов S и n и вычисляет n + 1 и S + $(n + 1)^2$ (см. объяснения в обзоре языка 8.2. Функция exch используется вместо функции swap языка Forth, а 1 index заменяет over из Forth).

Строка 9. В результате выполнения функции dosum с параметрами 3 и 6 получается 4 и 22. Создаются копии целых чисел 4 и 22, и число 22 печатается. Затем печатается строка из трех пробелов, и в текущей позиции курсора печатается 4.

Строка 10. Стек очищается от оставшихся в нем значений 4 и 22.

Строка 11. Курсор перемещается примерно на треть дюйма (25 точек).

Строка 12. В результате выполнения операций 0 1 9 {pop dosum} for числа 0, 1, ..., 9 будут последовательно размещаться в вершине стека с последующим выполнением процедуры {pop dosum}. На каждой итерации цикла вызываются функции pop (чтобы избежать использования индексов) и dosum, при этом стек операндов последовательно изменяется: (0, 0) ⇒ (1, 1) ⇒ (2, 5) ⇒ (3, 14) ⇒ ⇒ (4, 30) ⇒ ... (10,385). Затем функция formatit помещает верхний элемент стека (385) на страницу.

Строка 13. Полученная страница распечатывается.

Примеры операций Postscript. Для каждой операции указывается число необходимых для ее выполнения параметров (верхние элементы стека) и число вычисляемых значений, помещаемых в стек. Для обычной бинарной операции (например, add), которая получает два параметра из стека и заменяет их одним результирующим значением, задание параметров будет выглядеть как (2.1).

add(2.1)	Использует два верхних элемента стека в качестве параметров и заменяет их суммой их значений
aload(1.n+1)	Помещает элементы параметра-массива в стек ниже массива
array(1.1)	Создает массив длиной, равной значению параметра
astore(n+1.1)	Сохраняет n элементов в параметре-массиве, который является верхним элементом стека
Boolean(2.1)	Сравнивает два верхних элемента стека и помещает в вершину стека логическую величину (true или false). Boolean может принимать значения eq, ne, ge, gt, le или lt. Кроме того, можно использовать операции and, or, not и xor в их обычном значении
clear(n.0)	Извлекает все элементы из стека, очищая его
cleartomark(n.0)	Извлекает n элементов стека, начиная с верхнего
copy(n+1.2n)	n верхних элементов стека дублируются и располагаются в стеке
counttomark(n.n+1)	Пересчитываются элементы стека, начиная с указанного, и полученное значение помещается в стек
currentfont(0.1)	Текущий шрифт помещается в стек. (Эту операцию можно использовать для изменения размера текущего шрифта, даже не зная, какой он. Например, currentfont 2 scalefont setfont в 2 раза увеличивает размер текущего шрифта и устанавливает полученный шрифт в качестве текущего)
cvrs(3.1)	a b c cvrs преобразует число a в систему счисления по основанию b, сохраняет полученное значение в строке c и возвращает подстроку, содержащую a. Так, 123 10 10 string cvrs возвратит строку (123). 123 2 10 string cvrs преобразует число 123 в двоичную систему и возвратит это число как последовательность двоичных цифр
def(2.0)	/a b def создаст в пользовательском словаре userdict элемент с именем a, определяющий функцию по ее описанию b. Обычно b задается как список операций {x y z}
div(2.1)	div заменяет два верхних элемента стека на их частное

dup(1.2)	dup дублирует значение верхнего элемента стека, добавляя его в вершину стека
exch(2.2)	Меняет местами два верхних элемента стека
findfont(1.1)	По указанному имени шрифта из системного словаря помещает его в вершину стека
for(4.a)	a b c proc for поочередно помещает a, a + b, a + 2b ... в вершину стека до тех пор, пока не будет достигнуто значение c, вызывая на каждом шаге процедуру proc. Индекс a остается в стеке, если только процедура proc не удалит его оттуда
forall(2.a)	s proc forall вызывает процедуру proc для каждого элемента строки s
if(2.0)	b proc if выполняет процедуру proc, если b имеет значение true
ifelse(3.0)	b proc1 proc2 ifelse выполняет процедуру proc1, если b имеет значение true, и proc2, если b имеет значение false
index(n+2.n+3)	Последовательность a b c ... d n index берет *n*-й элемент стека и дублирует его в вершине стека. n=0 соответствует верхнему элементу стека в текущий момент, n=1 соответствует предыдущему элементу, и т. д. Операцию dup можно заменить операцией 0 index
length(1.2)	Помещает в вершину стека число элементов параметра-массива
mark(0.1)	Помещает в стек метку
moveto(2.0)	Два параметра этой операции представляют собой координаты x и y относительно левого нижнего угла страницы. Расстояние измеряется в точках (в дюйме содержится 72 точки)
mul(2.1)	mul заменяет два верхних элемента на произведение их значений
pop(1.0)	Удаляет верхний элемент стека
quit(0.0)	Завершает выполнение Postscript
repeat(2.1)	n proc repeat повторяет n раз процедуру proc
rmoveto(2.0)	Устанавливает новое положение курсора относительно текущей позиции
roll(n+2.n)	a b c ... d m n roll вызывает циклический сдвиг m объектов на n позиций
scalefont(2.1)	Операция заменяет существующий шрифт на масштабированный в соответствии с заданным масштабным множителем
setfont(1.0)	Устанавливает масштабированный шрифт как текущий
show(1.0)	Располагает верхний элемент стека на странице
showpage(0.0)	Печатает текущую страницу и затем очищает ее
string(1.1)	Создает строку, состоящую из нулей, длина строки определяется параметром операции (то есть 10 string создает строку из 10 нулей)
sub(2.1)	sub заменяет два верхних элемента стека разностью их значений

Операции среды. Эти операции сохраняют или восстанавливают состояние интерпретатора Postscript:

- save(0.1) и restore(1.0) — операция save помещает копию текущей среды в стек, а restore восстанавливает сохраненную среду. Эти операции оказываются полезными, если, например, к среде применяется масштабирование и преобразование, а потом требуется вернуться к исходной среде. Эти команды также сохраняют графическое состояние.
- gsave(0.1) и grestore (1.0) — операция gsave сохраняет текущее графическое состояние, а grestore восстанавливает его.

П.10.2. Графические команды

Эти команды используются для создания рисунков на странице.

arc(5.0)	x y r a b arc вычерчивает дугу окружности с центром в точке с координатами x и y и радиусом r от угла a (в градусах) до угла b. Например, чтобы нарисовать окружность с центром в точке с координатами (x, y), нужна следующая команда: x y moveto z y r 0 360 arc где z = x + r
closepath(0.0)	Замыкает контур, проводя линию к точке, которая указана командой newpath
currentpoint(0.2)	Помещает координаты текущей точки в стек
eofill(0.0)	Закрашивает область, ограниченную текущим контуром, цветом заливки. Для очистки области можно воспользоваться этой командой, установив белый цвет в качестве текущего. Для объектов неправильной формы используется правило четности и нечетности
erasepage(0.0)	Закрашивает страницу белым цветом (то есть стирает все содержимое страницы)
fill(0.0)	Закрашивает область контура цветом заливки. Для того чтобы стереть часть страницы, ее можно закрасить белым цветом с помощью этой команды
lineto(2.0)	Проводит линию из предыдущей точки в точку, координатами которой являются параметры этого оператора
newpath(0.0)	Начинает новый контур с текущий точки
scale(2.0)	Масштабирует систему координат, умножая координаты x и y этой системы на соответствующие параметры операции (множители)
setdash(2.0)	a b setdash устанавливает шаблон для вычерчивания линии. a — это массив, элементы которого извлекаются циклически и определяют последовательность отрезков и пробелов в линии. b указывает расстояние в массиве, откуда следует начинать, измеряемое в количествах сегментов от начала линии, а не как значение индекса массива. Поясним это на примерах: [] 0 setdash — сплошная линия [1]0 setdash — отрезок, пробел, отрезок... [1]1 setdash — пробел, отрезок, пробел... [123]2 setdash — пробел, 3 отрезка, пробел, 2 отрезка, 3 пробела...
setgray(1.0)	Устанавливает оттенок серого в качестве текущего цвета. 0 — соответствует черному, 1 — белому
setlinecap(1.0)	Определяет способ соединения линий, образующих фигуру: 0 встык, 1 — с закруглением, 2 — соединение под прямым углом
setlinewidth(1.0)	Устанавливает текущую толщину линии
stroke(0.0)	Рисует линию вокруг текущего контура
translate(2.0)	Сдвиг начала координат на расстояния, указанные в качестве параметров операции. Тем самым начало координат переносится из нижнего левого угла в новую точку

П.11. Prolog

Пример с пояснениями

В этом примере, как и в предыдущих случаях, мы приводим программу, которая суммирует элементы числового вектора. Для простоты мы предполагаем, что входные данные представлены в форме отношения Prolog: datavals(количество_элементов, [элементы_списка]). В листинге П.13 приведен пример выполнения такой программы.

Листинг П.13. Пример суммирования элементов массива на языке Prolog

```
 1 %editor data.prolog
 2     /* Данные считываются как отношения Prolog */
 3     datavals(4,[1,2,3,4]).
 4 %editor pgm.prolog
 5     go :- reconsult('data.prolog'),
 6        datavals(A,B),
 7        INSUM is 0,
 8        for(A,B,INSUM,OUTSUM),nl,
 9        write('SUM = '),write(OUTSUM),nl.
10     /* цикл for выполняется 'I' раз */
11     for(I,B,INSUM,OUTSUM) :- not(I=0),
12        B=[HEAD|TAIL],
13        write(HEAD),
14        NEWVAL is INSUM+HEAD,
15        for(I-1,TAIL,NEWVAL,OUTSUM).
16  /*Если I=0, возвращается вычисленное значение INSUM*/
17     for(_,_,INSUM,OUTSUM) :- OUTSUM = INSUM.
18     not(X) :- X, !, fail.
19     not(_).
20 %prolog
21 | ?- consult('pgm.prolog').
22 {consulting /aaron/mvz/book/pgm.prolog...}
23 {/aaron/mvz/book/pgm.prolog consulted, 30 msec 1456 bytes}
24 yes
25 | ?- go.
26 {consulting /aaron/mvz/book/data.prolog...}
27 {/aaron/mvz/book/data.prolog consulted, 10 msec 384 bytes}
28 1234
29 SUM = 10
30 yes
```

Строки 1–3. Вызывается редактор для создания файла данных. Вводом в нашей программе будет факт datavals, который содержит общее количество суммируемых чисел и список самих чисел.

Строка 4. Вызывается редактор для создания программы. Текст программы можно было бы ввести с клавиатуры непосредственно в Prolog с помощью встроенного предиката consult(user), вместо того чтобы заранее создавать его в файле, а потом вводить в Prolog с помощью предиката consult('pgm.prolog') в строке 21.

Строка 5. Здесь определяется основная цель программы — разрешение предиката go. Этот предикат буден разрешен, если разрешатся строки с 5 по 9. Сначала go модифицирует базу данных, считывая файл data.prolog и добавляя в базу данных любые новые факты, содержащиеся в этом файле, — в нашем случае это единственный факт datavals из строки 3.

Строка 6. Результатом выполнения этой строки будет присваивание (в действительности — унификация) A первого атома предиката datavals, количества суммируемых элементов (A=4), и B вектора элементов (B = [1,2,3,4]).

Строка 7. Значение INSUM устанавливается равным 0. is означает «требуется», = означает идентичность, и наш оператор просто унифицирует два объекта, чтобы они являлись одним и тем же, если это возможно.

Строка 8. В этой строке создается цикл for с использованием рекурсии. Правило будет выполняться A раз (A — это количество элементов в векторе B), передавая каждый раз вектор B и INSUM и возвращая OUTSUM = INSUM + элемент B. Поскольку

правило for будет печатать значение каждого элемента вектора B, функция nl после выполнения цикла напечатает символ новой строки.

Строка 9. По возвращении из правила for переменная OUTSUM содержит требуемую сумму значений элементов массива.

Строка 10. Комментарии в Prolog подобны комментариям в C и могут располагаться в любом месте.

Строки 11–17. Здесь задается правило for. Семантика этого правила — выполнить правило I раз. Если I — не (not) 0, то правило продолжает выполняться; если оно не является логическим следствием программы (fails), то алгоритм отката (backtracking) языка Prolog выполнит второе правило for (строка 17).

Строка 15. Новая частичная сумма и укороченный вектор TAIL снова рекурсивно передаются правилу for, при этом значение счетчика уменьшено на 1.

Строка 17. Строка 17 содержит второе правило for, которое выполняется, если первое правило for не является логическим следствием программы. Это происходит в том случае, когда правило not(I=0) не является логическим следствием программы (то есть I есть 0). В этом случае первые два аргумента игнорируем, и унифицируем сумму значений элементов массива (OUTSUM) со значением, которое вычислено на данный момент (INSUM). Заметим, что если поменять местами порядок выполнения двух правил for (то есть строки с одиннадцатой по пятнадцатую поставили бы на место семнадцатой, а ее переместили бы на их место), то в любом случае второе правило выполнилось бы в первую очередь и программа не могла бы работать нужным образом.

Строки 18–19. Это стандартное определение not, как объяснено далее в тексте.

Строка 20. Теперь, когда данные и программа созданы, вызывается Prolog.

Строка 21. Из файла pgm.prolog считываются правила. Можно было бы ввести предикат consult(user) и после этого непосредственно ввести правила в программу.

Строки 22–24. Это сообщение Prolog о том, что данные из файла были успешно прочитаны.

Строка 25. Ввод пользователем go. означает запрос, определенный в строках 5–9, который определяет успех программы.

Строки 26–27. Вывод из Prolog как результат чтения файла данных в строке 6.

Строки 28–29. Требуемый вывод из программы.

Строка 30. Prolog сообщает, что заданная в строке 25 основная цель go выведена. Выходом из Prolog часто является индикатор конца файла (например, <Ctrl+D>).

П.11.1. Объекты данных

Элементарные типы данных

Переменные и константы. Данные языка Prolog могут быть целыми числами (1, 2, 3, ...), вещественными числами (1.2, 3.4, 5.4, ...) и символами ('a', 'b', ...). Имена, начинающиеся со строчных букв, представляют конкретные факты, как если бы они являлись частью фактов из базы данных, а начинающиеся с прописных букв представляют переменные, которые могут быть унифицированы с помощью фактов во время выполнения программы. Областью видимости переменной является правило, которое ее использует.

Структурированные типы данных

Объекты могут быть *атомами* (строковые константы и переменные) или списками. Список представляется в виде [A. B. C. ...]. Запись [A|B] указывает на то, что A является головой списка, а B — хвостом:

```
?- X =[A|B]. X=[1.2.3.4].          - Запрос на 2 определения X
                                   - печать результатов унификации A. B и X
A = 1.
B = [2.3.4].
X = [1.2.3.4]
```

Строка 'abcd' представляет список ['a'.'b'.'c'.'d'].

Переменная _ представляет любую неименованную переменную. Так, запись [A|_] сопоставляет A с головой любого списка.

Определяемые пользователем типы

В Prolog нет реальных определяемых пользователем типов, однако правила, определяющие отношения, могут рассматриваться, как если бы они были пользовательскими типами.

Представление объектов в памяти

Правила и факты хранятся в памяти как связанные списки. Основная концепция выполнения программы на Prolog аналогична концепции LISP — обход дерева структуры связанных списков, описанной ранее при обсуждении LISP.

П.11.2. Управление последовательностью действий

Получив запрос, Prolog использует в качестве механизма управления памятью унификацию с откатом, как описано в разделах 8.4.3 и 8.4.4. В Prolog нет фактической концепции локальных или глобальных сред. Все правила имеют локальный контекст.

Способ реализации унификации определяет то, что запрос вида

$q_1. q_2. ... .q_n$

сначала вычисляет q_1, затем q_2 и т. д. Это выглядит таким образом, что любое правило выполняется последовательно. Кроме того, правила хранятся в том порядке, в котором они были внесены в базу данных. Таким образом, правило not скорее следует определять как

```
not(X) :- X. !. fail.
not(_).
```

а не как

```
not(_).
not(X) :- X. !. fail.
```

В последнем случае not(_) проверялось бы первым и всегда выводилось бы.

Выражения

В Prolog определены арифметические операции: +, -, *, mod и /. Также определены операции отношения =, =< (заметим, что запись этой операции отличается от обще-

принятой <=), <, > и >=. Но операция = означает «то же самое», так, X = 1 + 2 означает, что X есть то же самое, что и выражение 1 + 2. Операция is означает вычислить; результатом выполнения X is 4+1 будет присваивание переменной X значения 5:

```
|?- X = 1+2.
X = 1+2 ? - X присвоено выражение '1+2'
yes
|?- X is 1+2.
X = 3 ? - X присвоено значение 3
yes
```

Из-за такого смысла операции = выражение 1 + 3 = 2 + 2 будет ложным. Чтобы вычислять подобные сравнения, необходимо определить функцию равенства samevalue, которая использует конструкцию is для того, чтобы вычислить оба члена выражения:

```
samevalue(X.Y) :- A is X. B is Y. A=B.
```

Тогда samevalue(1 + 4.2 + 3) возвратит значение true.

Операторы

Факты. Факты — это отношения, которые устанавливаются в процессе выполнения операции consult, которая добавляет новые факты и правила в базу данных. Они оформлены в виде *n*-кортежа f(a_1. a_2,.... a_n) и устанавливают отношение между n аргументами в соответствии с отношением f. В ответ на ввод функции, consult(user), которая добавляет данные к базе данных, пользователь может ввести

```
employer(zelkowitz.university).
employer(smith.nasa).
employer(pratt.nasa).
publisher(zelkowitz.prenticehall).
publisher(pratt.prenticehall).

<Ctrl>+<D>
```

Эти отношения инвариантны. Они не могут быть унифицированы к некоторому другому значению.

Правила. Правила — это импликации, которые задаются во время операции consult. Синтаксис для правил следующий:

```
employed(X) :-employer(X._). /*X работающий, если у X имеется работодатель*/
employedby(X) :-employer(X.Y). write(Y).
```

Правило employed только проверяет, что в базе данных имеется отношение-предикат employer(X. что-то), таким образом, конкретный работодатель игнорируется. Но правило employedby(X) использует отношение employer(X.Y) для отображения имени работодателя:

```
employed(pratt)
yes
employedby(zelkowitz)
university
yes
```

Запросы. Запросы состоят из последовательности термов, завершающейся точкой:

q_1. q_2.....q_n.

Мы хотим, чтобы ассоциация для переменных в каждом терме была такой, чтобы все термы были истинными. Запрос выполняется следующим образом: унифи-

цируется терм q_1 (возможно, путем обращения к определению q_1 в базе данных); если он истинен, то терм q_2 унифицируется с помощью своего определения. Если в какой-либо точке процесс унификации сообщает об отказе, то он возвращается к предыдущему правильному выбору и пробует альтернативный путь, как описано в разделе 8.4.3.

Правила могут быть скомбинированы в сложные запросы. Например, вопрос «Кто работает в NASA и написал книгу, которая издана в „Прентис-Холл"» может быть выражен с помощью запроса:

```
employer(X.nasa).publisher(X.prenticehall).
X = pratt?
yes
```

Хотя и zelkowitz, и pratt унифицируют предикат publisher(X.prenticehall), только pratt унифицирует предикат employer(X.nasa).

Отсечение. Процесс вычисления запросов часто требует выполнения значительного числа откатов. Для экономии времени было предложено *отсечение*. Отсечение (обозначается символом «!») предписывает, что если в запросе необходимо применить откат, то мы получаем отказ в данном запросе. Действие отсечения заключается в том, чтобы ограничить пространство поиска для нахождения решения. Использование отсечения никогда не добавит решения, которое могло бы быть найдено без отсечения, но может исключить некоторые верные решения.

Например, если бы предыдущий запрос был записан в виде

```
employer(X.nasa).!.publisher(X.prenticehall).
```

тогда мы получили бы для него отказ. Сначала был бы унифицирован предикат employer(X.nasa), но получили бы отказ при унификации предиката издатель(X.Прентис-Холл) атомом smith. Обычно в такой ситуации Prolog выполнил бы откат и начал поиск другой подстановки для унификации предиката employer(X.nasa), но отсечение запрещает это делать. Отсечения не имеют никакого действия в прямом направлении.

Проблемы с отрицанием. Отрицание определяется следующим образом:

```
not(X):- X.!.fail.
not(_).
```

Заметим, это не соответствует тому, что следует *возвратить истину, если* X *ложно*. Если X истинно, то not(X) вычислит X как истина, откажет на fail, но отсечение приведет к отказу этого правила. Если X ложно, то при вычислении первого правила получим отказ, но not(_)выполнится успешно. Однако при вычислении первого правила можно получить отказ, если X ложно или просто отсутствует в базе данных. Разницу можно увидеть на примере следующих двух запросов:

```
X is 5. not(X=10).
not(X=10).X is 5
```

В первом случае X унифицируется целым 5 и not(X=10) выполнится успешно. Во втором случае X сначала унифицируется целым 10 и возникает отказ при вычислении not(X=10), так что унификация целым 5 никогда не выполняется.

Ввод и вывод

Для большинства простых запросов, чтобы записать ответы, вполне достаточно вывода по умолчанию унифицированных переменных. Тем не менее в Prolog су-

ществует также функция write для вывода на печать любой строки. Например, write('abc') напечатает abc. Функция nl печатает символ новой строки.

Стандартные функции

В большинство реализаций языка Prolog включены следующие встроенные функции, которые помогают генерировать программы:

consult(имя_файла)	Читает файл с заданным именем и добавляет новые факты и правила в базу данных. имя_файла может быть записано также в виде 'имя_файла', если имя файла содержит символы, не используемые в идентификаторах. consult(имя_файла) часто может быть заменено просто на [имя_файла]
reconsult(имя_файла)	Переписывает отношения в базе данных
fail	Всегда означает отказ
see(имя_файла)	Считывает входные данные из указанного файла в виде наборов правил
write(терм)	Выводит на печать терм
tell(имя_файла)	Направляет вывод функции write в файл с заданным имеием
told	Закрывает файл, использовавшийся в функции tell, и перенаправляет вывод функции write на стандартное устройство вывода (терминал)
nl	Переходит на новую строку (при вводе и выводе)
atom(X)	Предикат, который возвращает истину, если X является атомом (строковой константой или именем переменной)
var(X)	Предикат, который возвращает истину, если X является переменной
integer(X)	Предикат, который возвращает истину, если X является целым числом
trace	Включает отладочный режим трассировки программы, показывая выполнение каждого шага программы
notrace	Выключает отладочный режим трассировки программы

П.12. Smalltalk

Пример с пояснениями

Мы возвращаемся к нашему примеру суммирования элементов массива. В этой версии данные хранятся в файле data. Для входных данных

```
412345123450
```

программа, приведенная в листинге П.14, даст следующий результат:

```
1 2 3 4 SUM =10
1 2 3 4 5 SUM =15
```

Листинг П.14. Пример суммирования элементов массива на языке Smalltalk

```
1 Array variableSubclass: #Datastore
2     instanceVariableNames: ''
3     classVariableNames: 'DataFile ArrIndex Storage Size'
4     poolDictionaries: ''
5     category: nil !
6 !Datastore class methodsFor: 'instance creation'!
7 new
8     DataFile _ FileStream open:'data' mode: 'r'.
9     Storage _ Array new: 99.
10    Size _ 99.
```

продолжение

Листинг П.14 (*продолжение*)

```
11     self reset !!
12 !Datastore class methodsFor: 'basic'!
13 asgn: aValue
14     ArrIndex _ ArrIndex + 1.
15     Storage at: ArrIndex put: aValue !
16 getval
17     ArrIndex _ ArrIndex + 1.
18     ^Storage at: ArrIndex !
19 nextval
20     ^((DataFile next) digitValue - $0 digitValue)!
21 reset
22     ArrIndex _ 0 !!
23 |k j sum|
24 Datastore new.
25 "Initialize k"
26 [(k _ Datastore nextval) > 0]
27     whileTrue: [1 to: k do: [(j _ Datastore nextval) print.
28                              Character space print.
29                              Datastore asgn: j].
30              Datastore reset.
31              sum _ 0.
32              'SUM =' print.
33              1 to: k do: [sum _ sum + Datastore getval].
34              sum printNl.
35              Datastore reset] !
```

Строка 1. Здесь определяется Datastore как подкласс класса Array.

Строка 2. Пустой список указывает, что в любом объекте класса Datastore отсутствуют локальные переменные. В данном примере все данные обрабатываются в определении класса. Вообще говоря, каждый экземпляр класса имеет свой набор локальных переменных.

Строка 3. Переменные DataFile, ArrIndex, Storage и Size являются глобальными переменными объектов класса Datastore.

Строки 4–5. Эти строки нужны для корректного определения ключевого метода, но их обсуждение выходит за рамки данного примера. Символ (!) означает конец определения подкласса.

Строка 6. Здесь указано начало определения методов для класса Datastore, то есть вызовов методов вида

```
Datastore имя_метода
```

Строка 7. Определяется метод new. Этот метод открывает файл входных данных и инициализирует массив, в котором будут храниться данные.

Строка 8. Файл data открывается в режиме 'r' для чтения. Класс FileStream возвращает объект-дескриптор для этого файла, который присваивается переменной DataFile — глобальной переменной в определении подкласса Datastore в строке 3.

Строка 9. Метод new: посылается классу Array с параметром 99. Возвращается массив из 99 элементов, который присваивается переменной Storage, объявленной в строке 3.

Строка 11. Метод reset посылается объекту, представленному переменной self, который является объектом, куда изначально посылался метод new. Это приводит

к вызову метода reset для класса Datastore, определенному в строке 21. Два символа (!) указывают завершение определения вызываемых методов для класса Datastore из строки 6.

Строка 12. Для класса Datastore определяются основные операционные методы.

Строки 16–18. Унарный метод getval увеличивает значение переменной ArrIndex, представляющей индекс массива, на единицу и возвращает соответствующий элемент массива (^), используя метод at: для экземпляров массивов, в данном случае для массива Storage.

Строки 19–20. Метод nextval для файлового объекта DataFile возвращает следующий символ из файла. Метод digitValue, переданный этому символу, возвращает целое значение этого ASCII-символа. Вычитая из этого числа значение символа 0, получаем правильное численное значение для цифры целого числа (например, Character 0 ($0) возвратит целое число 0, Character 1 ($1) возвратит целое число 1 и т. д.).

Строка 23. k, j и sum определяются как локальные переменные. Все переменные объявляются без указания их типа, так как типы присваиваются динамически.

Строка 24. Вызывается метод new класса DataStore, объявленный в строке 7, для того чтобы открыть файл ввода и разместить массив данных.

Строка 25. Это комментарий Smalltalk.

Строка 26. Классу Datastore передается метод nextval, который возвращает следующее целочисленное значение из файла. Этот объект присваивается переменной k и сравнивается со значением 0. Результатом выполнения этого блока является либо булев объект true, либо булев объект false.

Строки 27–29. Метод whileTrue: с параметром-блоком передается булеву объекту строки 26. Если этот объект является объектом false, то блок игнорируется; если true, то блок выполняется и условный блок строки 26 вычисляется снова. Таким образом, реализуется стандартный цикл while.

Фактически блок представляет собой конструкцию цикла do. Метод to:do: получает в качестве параметров счетчик k и блок операторов и передает их целочисленному объекту 1. Результатом является циклическое выполнение блока от 1 до k.

Метод Datastore nextval возвращает следующий символ из файла, присваивает его переменной j и затем распечатывает значение j. Точка служит для разделения операторов.

Строки 30–32. Вызывается метод reset класса Datastore и переменной sum присваивается значение 0. Распечатывается строка 'SUM ='.

Строка 34. Распечатывается значение sum. После этого метод printNl вставляет символ новой строки.

Строка 35. Массив переводится в исходное состояние для нового суммирования. Символ (!) означает выполнение операторов, расположенных в строках 23–34.

П.12.1. Объекты данных

Переменные представляются цепочками строчных букв. В Smalltalk используется соглашение, по которому имена переменных могут состоять из нескольких слов, причем каждое следующее слово начинается с прописной буквы, например anInteger, aParameter, theInput, myInputFile и т. д. Имена глобальных объектов дан-

ных начинаются с прописной буквы. Все данные, которые совместно используются объектами (то есть объявленные в определении класса объекта), начинаются с прописных букв. Строки заключаются в одинарные кавычки (апострофы), а символьные данные начинаются с символа $. Так, например, символ "a" задается как $a. Выполнение программ является динамическим. Поэтому типы переменных не задаются, а устанавливаются, изменяются и запрашиваются во время выполнения.

Элементарные типы данных

В Smalltalk классы определяют типы. Множество предопределенных классов перечислено в табл. П.2.

Таблица П.2. Иерархия стандартных классов Smalltalk

Class	Object **(суперкласс — продолжение)**
Object	Delay
Autoload	FileSegment
Behavior	Link
ClassDescription	Process
Class	SymLink
Metaclass	Magnitude
BlockContext	Character
Boolean	Date
False	LookupKey
True	Association
CfunctionDescriptor	Number
Cobject	Float
Collection	Integer
Bag	Time
Mapped Collection	Memory
SequenceableCollection	ByteMemory
ArrayedCollection	WordMemory
Array	Message
ByteArray	MethodContext
CompiledMethod	MethodInfo
String	ProcessorSheduler
Symbol	SharedQueue
Interval	Stream
LinkedList	PositionableStream
Semaphore	ReadStream
OrderedCollection	WriteStream
SortedCollection	ReadWriteStream
Set	FileStream
Dictionary	Random
IdentityDictionary	TokenStream
SystemDictionary	UndefinedObject

Структурированные типы данных

В Smalltalk не имеется встроенных структурированных данных. Объекты данных создаются методами из определений классов.

Массивы. Массивы создаются при помощи ключевого метода new: класса Array, причем параметром метода является размер массива. Значения элементов массива задаются ключевым методом at:, как описано в разделе 7.2.5:

```
arrayVariable at:index put:arrayvalue
```

Метод at: используется также для получения значений элемента массива:

```
arrayVariable at:index
```

Множества. Множества создаются методом new: класса Set. Ключевой метод add: помещает в множество новый объект, метод remove: удаляет объект из множества, а метод includes: возвращает true или false в зависимости от того, входит ли параметр метода в данное множество.

Другие структуры. В Smalltalk имеются стандартные определения классов для связанных списков, множеств с повторяющимися элементами и других классов, перечисленных в табл. П.2. В вашей локальной системе могут быть и другие стандартные определения классов. Для того чтобы узнать атрибуты каждого такого класса, можно использовать метод inspect.

Определяемые пользователем типы

В Smalltalk полиморфизм и возможность задания пользовательских функций обеспечиваются наследованием методов в определениях классов.

Представление объектов в памяти

В Smalltalk используется динамическое управление памятью и указатели. Каждая переменная имеет указатель на соответствующий контекст для объектов этого класса.

П.12.2. Управление последовательностью действий

Выражения выполняются в порядке их задания. Операторы отделяются друг от друга точками. Определения методов завершаются символом «!», а определения классов завершаются символом «!!».

Выражения

Выражения в Smalltalk чрезвычайно просты. Существует только три уровня старшинства:

- унарный метод;
- бинарный метод;
- ключевой метод.

В пределах каждого уровня выполнение методов осуществляется слева направо. Например, выражение 2 + 3 * 4 содержит два бинарных метода с одинаковым старшинством, поэтому правильной его интерпретацией будет ((2 + 3) * 4) или

20, а не 14, как можно было ожидать. Возможно, это наиболее необычное свойство языка, и в этом отношении следует проявлять особую осторожность.

Также следует внимательно относиться к таким выражениям, как 1 + 2 print. Так как print — это унарная операция, то при расстановке приоритетов получим 1 + (2 print), то есть, вероятно, не то, что подразумевалось. Для того чтобы безошибочно получить желаемое семантическое значение выражения, всегда можно использовать скобки.

В Smalltalk определен обычный набор методов сравнения (например, <, <=, =, >, >=). Метод notequal (не равно) обозначается символами ~=.

Также определены некоторые дополнительные методы вычислений. Наиболее полезны из них следующие:

- // — частное (целочисленное деление);
- \\ — остаток;
- gcd: — наибольший общий делитель (то есть m gcd:n для целых чисел m и n).

Операторы

Объявления. Синтаксис |variable| определяет локальную переменную, область видимости которой распространяется до конца определения текущего метода (которое завершается символом «!»).

Присваивание. Основной оператор присваивания имеет вид

```
переменная _ выражение
```

В данном случае вычисляется выражение и полученное значение присваивается переменной. Выражение может быть как обычными данными, так и выполняемым блоком — таким образом конструируется оператор if. В GNU Smalltalk можно кроме стандартного оператора присваивания «_» использовать и оператор :=.

Блок. Блок имеет следующий синтаксис

```
[ :локальная_переменная | список операторов]
```

где локальная_переменная — необязательный компонент, представляющий собой список из одной или нескольких локальных переменных, областью видимости которых является данный блок. Метод value, посланный блоку, инициирует его выполнение. Обычно блоки присваиваются некоторой переменной x и затем выполняются, посылая сообщение value соответствующему блоку (например, x value).

Условные операторы. Условные операторы создаются посылкой метода ifTrue:ifFalse: объектам true или false. Объект true будет выполнять блоковый параметр ifTrue, тогда как объект false будет выполнять блоковый параметр ifFalse:

```
Булево_выражение
        ifTrue: [блок]
        ifFalse: [блок]
```

Булево_выражение будет вычислено равным true или false. Это значение будет передано ключевому сообщению ifTrue:ifFalse, после чего выполнится соответствующее действие.

Операторы цикла. Аналогично условным операторам, для создания циклических конструкций можно использовать ключевые методы.

Простые итерации. Метод timesRepeat: передает целому числу некоторый блок, в результате чего этот блок выполняется указанное количество раз. Так,

```
count timesRepeat:блок
```

выполнит блок count раз.

Фиксированное повторение. Код Smalltalk

```
начальное_значение to конечное_значение by шаг do: [блок]
```

аналогичен циклу в языке FORTRAN:

```
DO I = начальное_значение, конечное_значение, шаг
список_операторов
END DO
```

Как и в языке FORTRAN, шаг не является обязательным параметром. Можно получить доступ к текущему значению индекса цикла, задавая локальную переменную в цикле:

```
начальное_значение to: конечное_значение by: шаг do: [:LoopVar | блок]
```

Цикл while. Конструкция, аналогичная оператору while других языков, может быть задана следующим образом:

```
[Булево выражение]
      whileTrue: [блок]
```

где блок будет выполняться, пока булево выражение остается истинным.

Ввод и вывод

Функции ввода и вывода обеспечиваются классами ReadStream, WriteStream и их подклассами.

Метод print можно послать любому классу для распечатки его значения. Метод printNL действует так же, как print, но еще и добавляет символ новой строки в конце вывода.

Метод fileIn: класса FileStream считывает и выполняет файл. Например, в результате действия

```
FileStream fileIn: имя_файла
```

читается файл с заданным именем и обрабатывается Smalltalk. Это позволяет читать и выполнять другие программы во время выполнения основной программы.

Метод open:mode: класса FileStream открывает файл. Так,

```
FileStream open:имя_файла mode:режим
```

возвращает дескриптор файла для файла с заданным именем, который открывается только для чтения (если параметр режим задан как 'r') или только для записи (если параметр режим задан как 'w').

Метод next, примененный к экземплярам класса FileStream, возвращает следующий символ файла. Если файл с дескриптором f был открыт с помощью приведенного выше метода open:mode:, то

```
f next
```

возвращает следующий символ из f.

Определения классов

Определения классов состоят из четырех компонентов:

1) имени класса;
2) суперкласса данного класса;
3) объявлений переменных, доступных экземплярам класса;
4) методов, используемых экземплярами класса.

Подклассы определяются при помощи следующего синтаксиса:

```
имяСуперкласса subclass: #имяНовогоКласса
      instanceVariableNames: переменныеЭкземпляра
      classVariableNames: переменныеКласса
      poolDictionaries: ' '
      category: nil !
```

где имяНовогоКласса — это имя нового подкласса; переменныеЭкземпляра — список всех переменных, для которых отводится место в памяти в каждом экземпляре нового объекта и которые, таким образом, формируют структуру нового подкласса; переменныеКласса — это список переменных, которые совместно используются во всех экземплярах класса. Компонент category определяется с целью поддержки документирования класса, чтобы общие методы можно было сгруппировать вместе; на семантику языка этот компонент класса не оказывает никакого влияния. Компонент poolDictionaries позволяет перечислить переменные, которые будут использоваться в специфицированных классах системы вне нормальной иерархии наследования.

Определение методов. Методы, которые определяются для экземпляров класса, добавляются к определению класса с помощью следующего синтаксиса:

```
!имяКласса methodsFor: 'usage'!
метод1
      список_операторов1 !
метод2
      список_операторов2 !
...
методn
      список_операторовn !!
```

где usage задает категорию определяемого метода.

Если метод должен создавать экземпляры класса и является частью определения класса, то его синтаксис имеет вид

```
!имяКласса class methodsFor: 'usage'!
```

Например, в листинге П.14 показано объявление методов для создания экземпляра класса DataStore и задания основных операций для этого класса.

Управление памятью

Локальная среда ссылок. Наследование объектов формирует базу для локальной среды ссылок. Если метод, переданный объекту, не найден в нем, то вызывается класс родительского объекта. Аналогично в пределах данного метода доступны только переменные, заданные в определении класса или локально объявленные в нем.

Доступ к экземпляру класса, которому был послан метод, осуществляется с помощью объекта self. Чтобы обратиться в методе к родительскому классу, следует сослаться на объект super, например:

```
new
      ^ super new
```

в результате чего метод new сначала вызывает метод new в родительском классе и затем возвратит его экземпляр. Это позволяет модифицировать семантику данного метода в подклассе, а также вызывать родительский метод.

Общая среда ссылок. Все имена классов являются глобальными, то есть известны во всей программе, поэтому объекты этих классов могут быть вызваны из

любого метода. Объект Smalltalk предоставляет глобальный словарь. С помощью метода at:put: можно инициализировать любой объект, задавая его глобальное имя. Например,

```
Smalltalk at:#x put:(Array new:20)
```

создает глобальный объект x, который инициализирован как массив, состоящий из 20 элементов.

Передача параметров. Все параметры передаются по значению, так как в динамической модели хранения данных любой параметр просто копируется в новое место памяти, а потом происходит уборка мусора, в процессе которой освобождается память.

Возврат из метода. С помощью синтаксиса ^object метод возвращает объект. Если символ ^ не задан, то возвращается объект, содержащий вызванный метод.

Стандартные функции

Много предопределенных функций в определениях предопределенных классов в умалчиваемой среде Smalltalk становятся доступными при первоначальном вызове последнего. Некоторые функции уже были нами описаны. Некоторые другие полезные функции представлены ниже.

Класс Character. Функция Character value:anInteger возвращает символ, представленный целочисленным кодом anInteger.

Функция x digitValue возвращает целочисленное значение символа x. Как и в языке С, при вычислении выражения ($2 digitValue) - ($0 digitValue) результат будет равен 2.

Функции вывода на печать. Функция print выводит на печать значение объекта. Функция printNl делает то же самое, но еще и добавляет в конец выведенного значения символ новой строки. Фактически функция print посылает сообщение printOn:имя_файла объекту self. Это позволяет любому классу модифицировать информацию о себе, которую он печатает:

```
printOn: stdOutput
        super printOn:stdOutput. "Стандартный вывод"
        ...! "Специальный вывод для обьекта этого класса"
```

Системные функции. Объект-словарь Smalltalk выполняет много системных функций. Вот некоторые из них.

Метод inspect печатает внутреннюю информацию о структуре любого объекта.

Функция quitPrimitive (то есть Smalltalk quitPrimitive) позволяет выйти из Smalltalk. Символ конца файла (часто <Ctrl>+<D>) также можно вводить с клавиатуры для завершения выполнения программы.

Функция system:command вызывает выполнение команды command операционной системой. Например, Smalltalk system:'ls' распечатает имена локальных файлов в системах UNIX.

П.13. Рекомендуемая литература

Соответствующая документация ANSI описывает стандартные определения для языков FORTRAN 90 [11], С [10] и COBOL [9].

Хорошая сводка возможностей FORTRAN 90 дается в [21]. Книга [49] является хорошим обзором языка С. Ранние этапы развития FORTRAN и COBOL описаны в материалах конференции ACM по истории языков программирования, проводившейся в 1978 г. [117], а развитие языка С рассматривалось на второй такой же конференции, прошедшей в 1993 г. [4].

История Pascal изложена самим Виртом в [119]. Ясное и понятное описание языка было опубликовано Иенсеном (Jensen) и Виртом в [59]; аннотированная версия стандарта содержит несколько хороших советов по использованию языка [70]. Дальнейшие критические замечания о Pascal можно найти в [116]. Сравнению языков С и Pascal посвящена статья [40].

На основе Pascal были разработаны языки системного программирования: параллельный Pascal [22] и Modula [118]. Тем не менее в данной области ни тот, ни другой не смогли выдержать конкуренции с языком С.

История и развитие языка Ada описывается в [120]. Книга [20] дает более полное описание программирования на Ada, чем это было возможно здесь, тогда как [58] описывает стандарт языка.

Разработка языка С++ на основе С описывается в [108]. Подробному описанию С++ посвящено несколько недавно вышедших книг [72, 88 и 109].

История Smalltalk описана в [63]. Более полное описание стандарта языка Smalltalk-80 дано в [44]. Альтернативная разработка объектно-ориентированного языка на примере языка Eiffel представлена в работе Бертрана Мейера (Bertrand Meyer) [82].

Язык LISP, к сожалению, не имеет стандартного описания. Исходная версия языка дается в легкой для чтения книге Маккарти (McCatrhy) [80]. Диалект Scheme рассматривается в [1]. Common LISP определен в книге Стила (Steele) [105], а в [45] вы найдете хорошее введение в Common LISP. Обзор полной превратностей истории этого языка приведен в [106].

Спецификация стандартного ML описана в [83], в то время как книга Ульмана (Ullman) [115] представляет ясное описание языка, полезное при его изучении. Создание компилятора AT&T Standard ML (который использовался для разработки примеров в нашей книге) описано в [12].

Интерес к функциональным языкам программирования заметно возрос в конце 70-х гг., после знаменитой лекции Бэкуса (Backus) по случаю получения премии Тьюринга, в которой он критиковал «узкое место» компьютеров с архитектурой фон Неймана, отражающееся в традиционных языках программирования [15]. В качестве альтернативы языку ML Дэвидом Тернером (David Turner) был предложен язык Miranda, свойства которого во многом были аналогичны свойствам ML. Miranda — полностью функциональный язык, в то время как в ML допускается присваивание.

Начальный этап развития логического программирования описан в [67], а история Prolog — в [30]. Использование языка рассматривается в [28 и 107].

Библиография

1. *Abelson, H., Sussman G.J., and Sussman J.*, Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA (1985).
2. *Abrams, M. D., and Zelkowitz M. V.*, Belief in Correctness, National Computer Security Conference, Baltimore, MD (October, 1994), 132–141.
3. *ACM* Computing Surveys: Special Issue on Programming Language Paradigms (21)3 (1989).
4. *ACM* History of Programming Languages Conference II, Cambridge, MA (April 1993)[SIGPLAN Notices (28)3 (March 1993)].
5. *Formal* Definition of the Ada Programming Language, Honeywell, Inc. (preliminary) (1980).
6. *Adams D.*, Hitchhiker's Guide to the Galaxy, (1979).
7. *Adobe* Systems Inc., Postscript Language Reference Manual, Addison-Wesley, Reading, MA (1990).
8. *Aho, A., Sethi R., and Ullman J. D.*, Compilers: Principles, Techniques and Tools, Addison-Wesley, Reading, MA (1988).
9. *ANSI*, American National Standard Programming Language COBOL, X3.23, ANSI, New York (1985).
10. *ANSI*, American National Standard Programming Language C, X3.159, ANSI, New York (1989).
11. *ANSI*, American National Standard Programming Language FORTRAN 90, X3.198, ANSI, New York (1990).
12. *Appel, A. W., and MacQueen D. B.*, Standard ML of New Jersey, in Third International Symp. on Programming Language Implementation and Logic programming, M. Wirsing (Ed.), Springer-Verlag, New York (1991).
13. *Atkinson, M. P., and Buneman O. P.*, Types and Persistence in Database Programming Languages, ACM Computing Surveys (19)2 (1987), 105–190.
14. *Backus, J.*, The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference, Information Processing, UNESCO, Paris (1960), 125–132.
15. *Backus, J.*, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, Comm. ACM (21)8 (1978), 613–641.
16. *Berners-Lee T.*, WWW: Past, present, and future. IEEE Computer (29)10 (1996) 69–77.

17. *Biermann, A.*, Great Ideas in Computer Science: A Gentle Introduction, MIT Press, Cambridge, MA (1990).
18. *Bird, R., and Wadler P.*, Introduction to Functional Programming, Prentice-Hall, Englewood Cliffs, NJ (1988).
19. *Böhm, C., andJacobini, G.*, Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules, Comm. ACM (9)5 (1966), 366–371.
20. *Booch, G.*, Software Engineering with Ada, Benjamin/Cummings, Menlo Park, CA (1983).
21. *Brainerd, W. S., Goldberg C. H., and Adams J. C.*, Programmer 's Guide to FORTRAN 90, McGraw-Hill, New York (1990).
22. *Brinch Hansen, P.*, The Programming Language Concurrent Pascal, IEEE Trans. on Soft. Engr. (1)2 (1975), 199–207.
23. *Brown, A., Earl A., and McDermid J.*, Software Engineering Environments, McGraw-Hill International, London (1992).
24. *Burgess, B., Ullah N., P. van Overen, and Ogden D.*, The PowerPC 603 Microprocessor, Comm. ACM (37)6 (1994), 34–42.
25. *Cardelli, L.*, Compiling a Functional Language, Symp. on LISP and Functional Programming, ACM (1984), 208–217.
26. *Chandy, K. M., and Kesselman C.*, Parallel Programming in 2001, IEEE Software (8)6 (1991), 11–20.
27. *Chomsky, N.*, On Certain Formal Properties of Grammars, Information and Control 2 (1959), 137–167.
28. *Clocksin, W. F., and Mellish C. S.*, Programming in Prolog, Springer-Verlag, Berlin (1987).
29. *Cohen, J.*, Garbage Collection of Linked Data Structures, ACM Computing Surveys (13)3 (1981), 341–368.
30. *Colmerauer, A., and Roussel P.*, The Birth of Prolog, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)] 37–52.
31. *Conway, R., and Maxwell W. L.*, CORC: The Cornell Computing Language, Comm. of the ACM (6)6 (1963), 317 –321.
32. *Conway, R. W., and Wilcox T.*, Design and Implementation of a Diagnostic Compiler for PL/I, Comm. ACM (16)3 (1973), 169–179.
33. *Dahl O., Dijkstra E., and Hoare C. A. R.*, Structured Programming, Academic Press, New York (1972).
34. *Davie B., Peterson L., Clark D.*, Computer Networks: A Systems Approach, Morgan Kaufmann, San Francisco, CA (1999).
35. *Dershem, H., and Jipping M.*, Programming Languages: Structures and Models, PWS Publishing Co., Boston (1995).
36. *Dijkstra, E.*, Notes on Structured Programming. In [33], 1–82.
37. *Dijkstra, E. W.*, Guarded Commands, Nondeterminacy, and Formal Derivation of Programs, Comm. ACM (18)8 (1975), 453–457.
38. *Duncan, R.*, A Survey of Parallel Computer Architectures, IEEE Computer (23)2 (1990), 5–16.

39. *Falkoff, A. D., and Iverson K. E.*, The Evolution of APL, ACM History of Programming Languages Conference, Los Angeles, CA (June 1978) [SIGPLAN Notices (13)8 (August 1978)], 47–57.
40. *Feuer, A., and Gehani N.*, A Comparison of the Programming Languages C and Pascal, ACM Computing Surveys (14)1 (1982), 73–92.
41. *Fischer, C. N., and LeBlanc R.*, Crafting a Compiler, Benjamin Cummings, Menlo Park, CA (1988).
42. *Furht, B.*, Parallel Computing: Glory and Collapse, IEEE Computer (27)11 (1994), 74–75.
43. *Gannon, J. D., Purtilo J. M., and Zelkowitz M. V.*, Software Specification: A Comparison of Formal Methods, Ablex, Norwood, NJ (1994).
44. *Goldberg, A.*, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, Reading, MA (1984).
45. *Graham, P.*, On Lisp: Advanced Techniques for Common Lisp, Prentice-Hall, Englewood Cliffs, NJ (1994).
46. *Griswold, R.*, String and List Processing in SNOBOL4: Techniques and Applications, Prentice-Hall, Englewood Cliffs, NJ (1975).
47. *Griswold, R. E.*, A History of the SNOBOL Programming Language, ACM History of Programming Languages Conference, Los Angeles, CA (June, 1978) [SIGPLAN Notices (13)8 (August 1978)], 275–308.
48. *Guttag, J. V.*, Notes on Type Abstraction (Version 2), IEEE Trans. on Software Engineering (6)1 (1980), 13–23.
49. *Harbison, S. P., and Steele G. L.*, C: A Reference Manual, 4th edition, Prentice-Hall, Upper Saddle River, NJ (1995).
50. *Hoare, C. A. R.*, Notes on Data Structuring. In [33], 83–174.
51. *Hoare, C. A. R., and Wirth N.*, An Axiomatic Definition of the Programming Language Pascal, Acta Informatica 2 (1973), 335--355.
52. *Hoare, C. A. R., and Lauer P. E.*, Consistent and Complementary Formal Theories of the Semantics of Programming Languages, Acta Informatica 3 (1974), 135–153.
53. *Horstmann C. S.*, Practical Object-Oriented Development in C++ and Java, John Wiley &Sons, New York (1997).
54. *IBM*, The Man Behind FORTRAN, IBM Computing Report (2)4 (November 1966), 7–10, 19.
55. *IEEE*, Binary Floating Point Arithmetic, Standard 754, IEEE, New York (1985).
56. *Ingalls, D. H.*, The Smalltalk-76 Programming System: Design and Implementation, ACM Symp. on the Principles of Programming Languages, (January 1978).
57. *Intermetrics*, Ada 95 Rationale, (January 1995).
58. *ISO*, Ada 95 Reference Manual, ISO/IEC 8652:1995, International Organization for Standardization/International Electrotechnical Committee (December, 1994).
59. *Jensen, K., and Wirth N.*, Pascal User Manual and Report, Springer-Verlag, New York (1974).
60. *Johnson, M., and Johnson W.*, Superscalar Microprocessor Design, Prentice-Hall, Upper Saddle River, NJ (1991).

61. *Jones, C. B.*, Systematic Software Development Using VDM, Prentice-Hall, Englewood Cliffs, NJ (1990).
62. *Kay, A.*, The Early History of Smalltalk, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 67–95.
63. *Kelly-Bootle, S.*, The Computer Contradictionary, MIT Press, Cambridge, MA (1995).
64. *Knuth, D. E.*, Semantics of Context-Free Languages, Mathematical Systems Theory 2 (1968), 127–145.
65. *Knuth, D.*, The Art of Computer Programming, Vols 1 to 3, Addison-Wesley, Reading, MA (1968, 1969, 1973).
66. *Knuth, D. E.*, The TEXbook, Addison-Wesley, Reading, MA (1984).
67. *Kowalski, R.*, The Early History of Logic Programming, Comm. ACM (31)1 (1988), 38–43.
68. *Kurtz, T. E.*, BASIC, ACM History of Programming Languages Conference, Los Angeles, CA (June 1978) [SIGPLAN Notices (13)8 (August 1978)], 103–118.
69. *Lamport, L.*, LaTeX: A Document Preparati on System, Addison-Wesley, Reading, MA (1986).
70. *Ledgard, H.*, The American Pascal Standard with Annotations, Springer-Verlag, New York (1984).
71. *Linger, R. C., Mills H. D., and Witt B. I.*, Structured Programming: Theory and Practice, Addison-Wesley, Reading, MA (1979).
72. *Lippman, S. B.*, C++ Primer, Third Edition, Addison-Wesley, Reading, MA (1998).
73. *Lopez, L. A., Valimohamed K. A., and Schub L. G.*, An Environment for Painless MIMD System Development, IEEE Software (9)6 (1992), 67–76.
74. *Louden, K.*, Programming Languages: Principles and Practice, PWS-Kent, Boston, MA (1993).
75. *Lucas, P., and Walk K.*, On the Formal Description of PL/I, Ann. Rev. Auto. Prog. (6)3 (1969), 105–182.
76. *Maddux, R.*, A Study of Program Structure, Ph.D. dissertation, University of Waterloo (July 1975).
77. *Mano, M., and Kime C.*, Logic and Computer Design Fundamentals, Prentice-Hall, Upper Saddle River, NJ (1999).
78. *Marcotty, M., Ledgard H., and Bochman G.*, A Sampler of Formal Definitions, ACM Computing Surveys (8)2 (1976), 191–275.
79. *Martin, A.*, Internationalization Explored, Uniforum, San Francisco, CA (1992).
80. *McCarthy, J.*, LISP 1.5 Programmer's Manual, 2nd edition, MIT Press, Cambridge, MA (1961).
81. *Metzner, J., and Barnes B.*, Decision Table Languages and Systems, Academic Press, New York (1977).
82. *Meyer, B.*, Eiffel: The Language Prentice-Hall, Upper Saddle River, NJ (1990).
83. *Milner, R., Tofte M., and Harper R.*, The Definition of Standard ML, MIT Press, Cambridge, MA (1989).

84. *Morgan, H.*, Spelling Correction in Systems Programs, Comm. ACM (13)2 (1970).
85. *Mulkers, A., Winsborough W., and Bruynooghe B.*, Live Structure Dataflow Analysis for Prolog, ACM Trans. on Prog. Lang. and Systems (16)2 (1994), 205–258.
86. *Parnas, D. L.*, On the Criteria to be Used in Decomposing Systems into Modules, Comm. ACM (15)12 (1972), 1053–1058.
87. *Perry, D., and Kaiser G.*, Models of Software Development Environments, IEEE Trans. on Soft. Eng. (17)3 (1991), 283–295.
88. *Pohl, I.*, C++ distilled: A Concise ANSI/ISO Reference and Style Guide, Addison-Wesley, Reading, MA (1996).
89. *Prenner, C., et al.*, An Implementation of Backtracking for Programming Languages, in Control Structures in Programming Languages (Special Issue), SIGPLAN Notices (7)11 (1972).
90. *Rada, R., and Berg J.*, Standards: Free or Sold?, Comm. ACM (38)2 (1995), 23–27.
91. *Rather, E., Colburn D., and Moore C.*, The Evolution of Forth, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 177–199.
92. *Ritchie, D. M.*, The Development of the C Language, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 201–208
93. *Rosen, S.*, Programming Systems and Languages, McGraw-Hill, New York (1967).
94. *Sajeev, A. S. M., and Hurst A. J.*, Programming Persistence in χ, IEEE Computer (25)9 (1992), 57–66.
95. *Sammet, J.*, Programming Languages: History and Fundamentals, Prentice-Hall, Englewood Cliffs, NJ (1969).
96. *Sammet, J.*, Programming Languages: History and Future, Comm. ACM (15)7 (1972), 601–610.
97. *Sammet, J. E.*, The Early History of COBOL, ACM History of Programming Languages Conference, Los Angeles, CA (June 1978) [SIGPLAN Notices (13)8 (August 1978)], 121–161.
98. *Scott, D.*, Lattice Theory, Data Types and Formal Semantics, Formal Semantics of Programming Languages [R. Rustin (Ed.)], Prentice-Hall, Englewood Cliffs, NJ (1972), 65–106.
99. *Sebesta, R.*, Concepts of Programming Languages, 4th edition, Benjamin Cummings, Redwood City, CA (1998).
100. *Sethi, R.*, Programming Languages: Concepts and Constructs, 2nd edition, Addison-Wesley, Reading, MA (1996).
101. *Shaffer C.*, A practical introduction to data structures and algorithm analysis, Prentice-Hall, Upper Saddle River, NJ (1996).
102. *Smith, J. E., and Weiss S.*, PowerPC 601 and Alpha 21064: A tale of Two RISCs, IEEE Computer (27)6 (1994), 46–58.
103. *Spivey, J. M.*, An Introduction to Z and Formal Specifications, Software Engineering Journal (4)1 (1989), 40–50.

104. *Stansifer, R.*, The Study of Programming Languages, Prentice-Hall, Englewood Cliffs, NJ (1995).
105. *Steele, G.*, Common LISP: The Language, 2nd edition, Digital Press, Bedford, MA (1990).
106. *Steele, G. L., and Gabriel R. P.*, The Evolution of LISP, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 231–270.
107. *Sterling, L., and Shapiro E.*, The Art of Prolog, MIT Press, Cambridge, MA (1986).
108. *Stroustrup, B.*, A History of C++: 1979–1991, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 271–297.
109. *Stroustrup, B.*, The C++ programming language, Special Edition, Addison-Wesley, Reading, MA (1990).
110. *Taivalsaari, A.*, On the Notion of Object, J. of Systems and Software, (21)1 (1993), 3–16.
111. *Tanenbaum, A.*, Structured Computer Organization, 3rd edition, Prentice-Hall, Englewood Cliffs, NJ (1990).
112. *Tanenbaum, A. S., Kaashoek M. F., and Bal H. E.*, Parallel Programming Using Shared Objects and Broadcasting, IEEE Computer (25)8 (1992), 10–19.
113. *Tucker, A. B. (Ed.)*, Computing Curricula 1991, Comm. ACM (34)6 (1991), 68–84.
114. *Turing, A.*, On Computable Numbers, with an Application to the Entscheidungs-Problem, Proc. London Math. Soc. 42 (1936), 230–265.
115. *Ullman, J. D.*, Elements of ML Programming, Prentice-Hall, Englewood Cliffs, NJ (1994).
116. *Welsch, J., Sneeringer W., and Hoare C. A. R.*, Ambiguities and Insecurities in Pascal, Software–Practice and Experience (7)6 (1977), 685–696.
117. *Wexelblat, R. (Ed.)*, History of Programming Languages, Academic Press, New York (1981).
118. *Wirth, N.*, Programming in Modula-2, Springer-Verlag, Berlin (1988).
119. *Wirth, N.*, Recollections About the Development of Pascal, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 333–342.
120. *Whitaker, W. A.*, The Ada Project: The DOD Higher Order Language Working Group, ACM History of Programming Languages Conference II, Cambridge, MA (April 1993) [SIGPLAN Notices (28)3 (March 1993)], 299–332.
121. *Wyatt, B, Kavi K., and Hufnagel S.*, Parallelism in Object-Oriented Languages: A Survey, IEEE Software (9)6 (1992), 56–65.
122. *Workman, D.*, ACM Conference on Language Design for Reliable Software, Raleigh, NC, SIGPLAN Notices (12)3 (March 1977).
123. *Zelkowitz, M. V.*, The Role of Verification in the Software Specification Process, Advances in Computers 36, Academic Press (1993), 43–109.

Алфавитный указатель

Символы

λ-выражения 161
  моделирование математики 163
  операция редукции 162
  способы передачи параметров 163
λ-исчисление 161

A

Acrobat 542
ACVS 500
Ada 499
  ввод-вывод 576
  выражения 573
  дискриминант 262
  задачи 502, 503, 577
  закрытые типы 579
  исключительные ситуации 577
  история 499
  массивы 569
  наследование 580
  обзор 501
  объекты данных 567
  оператор цикла 575
  операторы 574
  определяемые пользователем типы 570
  пакеты 305, 578
  передача по копии 420
  переменные 567
  подтипы 572
  помеченные типы 312
  пример программы 564
  производные типы 572
  рандеву 508
  реализация пакетов 305
  реализация параметров 422
  символьные строки 570
  символьный и булев типы 569
  указания компилятору 576
  указатели 569
  условные операторы 574
  файлы 570
  числовые данные 567
  элементарные типы 567
Ada 9X 501
Adams D. 530
Adobe 542
Adobe Systems 536
Aho, Alfred 128
AJPO 500
ALGOL 20
Anderssen M. 88
ANSI 48, 549
APL 341
  распределение памяти 481
Appel D. 169
Apple 26
ARPA 542
ARPANET 29, 543
ASCII 97
AT&T Bell Telephone
  Laboratories 58, 169, 293
AWK 128

B

B 58
B-news 128
Babbage C. 500
Backus J. 19
BASIC 26, 103, 104
BCPL 58
Berners-Lee T. 29, 545
BNF грамматика 21
Bohm C. 366
BSI 48
Burroughs 21

C

C 58
  struct 258
  блоки 587
  ввод и вывод 589
  выражения 586
  директивы препроцессора 588
  история 58
  массивы 584
  обзор 59
  объединение 585
  объекты данных 583
  оператор switch 588
  операторы 587

C (*продолжение*)
операторы цикла 587
операторы-выражения 587
организация памяти в виде кучи 222
передача управления 588
переменные 583
подтипы 191
представление в памяти 586
пример программы 581
приоритеты 340
присваивание 349
прототип 194
прототип функции 189
расширение 199
среда окружения 59
строковые данные 583
структуры 585
указатели 584
условные операторы 587
функции 590
целые числа 188
числовые типы данных 583
C++ 293
private 313
protected 315
public 313
ввод-вывод 599
виртуальная функция 318
виртуальные функции 597
выражения 598
дружественные классы 595
история 293
классы 594
макрос assert 488
обзор языка 293
операторы 599
определение класса 313
представление в памяти 598
пример программы 592
производные классы 313, 314, 597
указатели this 596
функции 601
шаблоны 597
CBEMA 48
CBL 22
CEP
указатель на текущую среду ссылок 388
CERN 29, 545
CGI-сценарий 554
Chomsky N. 21, 114, 142, 144
Church A. 150
CIP
указатель на текущую команду 387
CISC 67, 518
COBOL 354, 355
раздел оборудования 103
Common LISP 465
Coulmerauer A. 368

D

DARPA 28
Dijkstra E. 496
DOD-1 500
DTD 550, 561
Dynabook 319

E

ep указатель среды 388

F

FIXED DECIMAL 200
Floyd R. 143
Forth 538
FORTRAN 229
EQUIVALENCE 51
булев тип данных 604
ввод-вывод 609
выражение 607
история 230
история создания 19, 34
краткий обзор языка 230
массивы 605
объявление типов по умолчанию 193
операторы 607
переменные 603
пример программы 601
символьные строки 606
соглашение об именах 603
управление подпрограммами 609
управление последовательностью действий 607
числовые типы данных 604
FORTRAN 90
INTERFACE 284
интерфейсный блок 284
FORTRAN 90
сечения 255
forward
аномалия (Pascal) 396
объявление 395
Free Software Foundation 320
FTP 29

G

GAMM 20
Geshke C. 536
GIF 552
GML 549
Goldfarb C. 549
gopher 553
Gosling J. 87

H

Hoare C.A.R. 143, 158
Hollerith card 24
Hollerith G. 24
Hooper G. 19, 22
HotJava 88

HTML 29, 532, 548, 550
index.html 550
апплеты 553
протокол 552
рисунки 552
сценарии CGI 555
таблицы 553
формы 554
HTTP 29, 546
HTTPD 547
HyperCard 545

I

IBM 20, 34, 57
Ichbiah J. 500
IEEE 48
ip указатель команды 388
ISO 48
l-значение 202

J

Jacobini W. 366
Java 27, 87, 554, 558
private 313
абстрактные классы 615
атрибут static 614
безопасность 89
библиотека классов AWT 615
булевые данные 612
виртуальная машина 559
классы 613
конструктор 614
массив 313
массивы 612
метод finalize 614
обзор 87
операторы 613
определение метода 614
потоки 615
пример программы 611
производные классы 314
символьные данные 612
строки 613
субклассы 614
управление доступом 614
числовые данные 612
экземпляры 613
язык программирования 88
JCL 57
JPEG 552

K

Kay A. 319
Knuth D. 142, 159, 533
Kowalski R. 368

L

Lamport L. 533
LaTeX 533
структура 533
LISP 329, 465
CLOS 466
cons 265
join 267
атом 466
атомы 617
ввод и вывод 622
выражения 620
история 267, 465
история развития 22
обзор языка 465
определение функций 622
переменные 617
пример программы 616
списки 265
списки свойств 618
управление последовательностью действий 619
условные выражения 620
функции 623
числовые данные 618
элементы списка 329
Lorie R. 549
Lovelace A. 500
Lukasiewicz W. 21, 335

M

Macintosh 26
MacQueen D. 169
Maddux R. 363
mailto 553
Mascitti R. 293
McCarthy J. 22, 465
Microsoft RTF 532
Mills H. 158, 366
Milner R. 168
MIMD 521
MIT 22, 23, 24
mixin (смешанный) 319
ML 168
Standard ML 168
абстракции 633
ввод и вывод 632
выражение as 631
выражения 629
исключения 485, 632
история 168
массивы 635
наследование типов 198
обзор 168
объявление open 634
операторы 630
определение функций 630
определяемые пользователем типы 627
переменные 626
полиморфизм 328
пример программы 624
сопоставление с образцом 372
списки 265

ML (*продолжение*)
структурированные типы данных 626
структуры 633
функции 632
Mosaic 29, 87, 546
Mosher E. 549
Multics 24, 58

N

Naur P. 20
NCSA 546
Netscape 88
news 553
NIST 48

P

P-код 112
Pascal 135
P-код 112
textfile 224
булевы данные 638
вариантные записи 262
ввод и вывод 643
выражения 641
записи 639
история 135
история создания 21
массив 638
множества 639
обзор языка 136
объекты данных 637
объявление forward 395
оператор goto 643
операторы цикла 642
определяемые пользователем типы данных 641
переменные 637
пример программы 635
символьные данные 638
символьные строки 639
составные операторы 642
тег 262
указатели 638
управление кучей 481
условные операторы 642
файлы 640
функции 643, 644
числовые данные 637
PDF 542
Perl 127, 257
ассоциативные массивы 646
ассоциативный массив 257, 554
булевы данные 646
история 128
массивы 646
обзор 128
операторы 648
подпрограммы 649
подстановка 648
Perl (*продолжение*)
пример программы 645
регулярные выражения 647
строки 646
функции 649
характеристика языка 128
PL/C 200
PL/I 22
переполнение 201
преобразование типов 200
Postscript 532, 536
systemdict 537
userdict 537
закрашивание замкнутых фигур 540
имена 537
команды раскрашивания 538
литералы 538
массивы 650
модель выполнения 536
начало координат 538
операторы среды 652
процедуры 650
соглашения 536, 540
стек выполнения 537
стек графики 537
стек операндов 537
стек словарей 537
строки 650
точки 538
элементарные данные 649
private 614
Prolog 368
алгоритм обхода дерева 376
ввод и вывод 658
выполнение программы 376
выражения 657
запрос 373, 657
история создания 23
краткое описание 369
отрицание 658
отсечение 658
переменные 655
правило 373, 657
представление в памяти 656
пример программы 653
принцип резолюции 380
присваивание 204
равенство 204
структурированные типы данных 656
унификация 373, 375
управление последовательностью действий 656
факт 373, 657
функции 659
protected 315, 614
public 614

R

r-значение 202
Rice S. 549
RISC 67, 518

Ritchie D. 58
Robinson J. 380
Roussel P. 368
RPC (вызов удаленных процедур) 527
RTF 532

S

Sammet J. 16
Scharpf N. 549
Scheme 329
Scheme (версия LISP) 465
Schwartz J. 21
Scott D. 158
SED 128
self (Smalltalk) 324
SGML 548
  DTD 550
  история 548
  синтаксис 549
  тег 549
  элементы 549
shell 34, 57
Simula 88
Smalltalk 319
  self 324
  абстракция 323
  бинарное сообщение 322
  блок 664
  блоки операторов 322
  ввод и вывод 665
  выражения 663
  другие структуры 663
  история 319, 322
  класс 320
  ключевые сообщения 322
  комментарии 323
  массивы 663
  множества 663
  наследование классов 324
  обзор 320
  объекты данных 661
  операторы 664
  операторы цикла 664
  определение методов 666
  определение подкласса 324
  определения классов 665
  передача параметров 667
  передача сообщений 322
  представление в памяти 663
  пример программы 659
  присваивание 664
  создание объектов 321
  сообщение 320
  структурированные типы данных 663
  структуры управления 323
  унарное сообщение 322
  управление памятью 321, 666
  управление последовательностью действий 663
  условные операторы 664
  функции 667
  элементарные типы данных 662
SMTP 29, 545
SNOBOL 101
SNOBOL4 370
SRI 543
Standard ML 168
Standford University 136, 142
Steel G. 465
Strachey C. 158
Stroustrup B. 21, 88
Sun Microsystems 87, 554
Susmann G. 465
System 360 57
systemdict 537

T

TCP/IP 543
Teitelman W. 465
Telnet 543
telnet 29
TeX 33, 533
Thompson K. 58
Turing A. 149
typedef 286

U

UCLA 543
Unisys 21
Univac 19
UNIX 58, 59, 88
  shell 57
  исключение 487
URL 30, 546, 551
userdict 537

V

Vienna Definition Language 157
Visi-Calc 26

W

W3C 561
wait 506
Wall L. 128
Warnock J. 536
Web-браузер
  Mosaic 29
Weinberger P. 128
Windows 26
Wirth N. 21, 135, 293, 421
World Wide Web 29, 51
WWW 545
  HTML 550
  URL 551
  домашняя страница 546
WYSIWYG 535

X

Xerox PARC 319
XML 561

Y

YACC 133
Yngve V. 23

А

абсолютный адрес 222
абстрактные классы 318
абстрактные типы данных 272
  общие 308
  реализация 305
абстракция 274
  поддержка 275
абстракция данных 304
  инкапсуляция 273
агрегация 327
Ада
  пакеты 103
Адамс Д. 530
адресация 67
аксиоматическая верификация 171
  пример доказательства 173
аксиоматические модели 158
активация подпрограммы 387
  статическая цепь 446
алгебраические типы данных 158, 174
  аксиомы для стека 175
  генераторы 174
  индукция типов данных 176
  конструкторы 174
  формирование аксиом 174
  функции 174
алгоритм 189
алгоритм маркировка-сбор 472
алфавитный порядок 52
Андерсен М. 88
Аппель Д. 169
апплет 87, 554
аппликативная модель 157, 161
аппликативные языки 43
арность операции 190
архитектура программного обеспечения 523
архитектура фон Неймана 70
ассемблер 73
ассоциативный массив 128, 257
  значение 257
  ключ 257
ассоциация 399
атом 466, 617
атомарность 506
атрибутивная грамматика 142, 159

Б

базовый адрес 222
байт-коды 559
Бем К. 366
Бернерс-Ли Т. 29, 545
бинарная операция 189
блок 407
блок-схема 46
блоковый дескриптор 330
блочная структура 407
  дисплей 448
  локальные объявления 450
  реализация статической цепи 446
браузер 545
  HotJava 559
булевы значения 214
Бэббидж Ч. 500
Бэкус Дж. 19, 21, 34

В

вариантные записи 262
  реализация 263
Вейнбергер П. 128
вектор предварительной информации 188
векторы 246
  атрибуты 247
  границы диапазона 248
  операции 247
  реализация 248
  упакованное представление 250
  формула доступа 248
верификация 95
взаимное исключение 513
Вирт Н. 21, 135, 293, 421
виртуальная архитектура 73
виртуальная функция 318
  реализация 318
виртуальные машины 80
виртуальный компьютер 65, 73, 78
  иерархия 80
виртуальный начальный адрес 249
вложенные операторы 101
восстановление памяти 463
время жизни 182
время Интернета 58
время связывания 82
  значение 85
  позднее связывание 86
  при выполнении программы 82
  при определении языка 83
  при реализации языка 83
  при трансляции 83
  раннее связывание 86
Всемирная паутина 29, 51
вспомогательные объекты данных 182
встраиваемая последовательность кодов 192
встроенный компьютер 26
выборка компонента 238
вызов по имени 21
вызов удаленных процедур (RPC) 527
вызывающая последовательность 53
выражения 100
  выполнение 343
  вычисление 344
  вычисление по укороченной схеме 574

выражения (*продолжение*)
двучленные операции 336
неарифметические 367
побочные эффекты 346
правила приоритета 339
правило активного вычисления 344
правило ленивого вычисления 345
семантика 336
синтаксис 335
сопоставление с образцом 369
укороченное вычисление 347, 348
управление последовательностью действий 333
условие возникновения ошибки 347
выражения as (ML) 631
вычисления
унифицированные правила вычислений 344

Г

Гешке Ч. 536
гиперссылка 29
гипертекст 545
главный компьютер 543
голова списка 267
Гольдфарб Ч. 549
Гослинг Дж. 87
грамматика 112, 142
эквивалентность машине 153
грамматика с фразовой структурой 147
грамматические правила (НФБ-грамматика) 115
рекурсивное правило 117
группирование 327

Д

двучленные операции 336
Дейкстра И. 496
декомпозиция 327
демон 547
денотационная семантика 142, 158, 164
память 165
семантика операторов 167
состояние программы 165
дерево грамматического разбора 117
деревья 267
дескриптор 188, 192
детерминированный конечный автомат 124
динамическая область видимости 404, 439
правило последней ассоциации 440
динамическая проверка 263
дисплей 449
документация программы 532
доступ к массивам 248
древовидное представление 334
дружественный доступ 614

Е

единообразие понятий (свойство языка) 38

Ж

железнодорожная диаграмма 122

З

зависание программы 507
зависимость от предыстории 190
заглушка 54
загрузка файлов 30
загрузчик 74
задача производитель-потребитель 507
задачи 502
режим реального времени 510
синхронизация 504
управление задачами 502
управление памятью 511
закрытые объекты 305, 614
записи 257
атрибуты 258
вариантные 262
реализация 259, 261
синтаксис 257
запись активации 387
подпрограммы 282
зарезервированные слова 99
затравка 191
защищенный объект 315, 614
значение данных 182
значок 57

И

иерархия Хомского 144
избыточный синтаксис 95
имена (категории) 399
именованная константа 183
императивный язык 43
Ингве В. 23
индексно-последовательный файл 229
индексы 256
индивидуализация 327
индукция типов данных 175
инкапсуляция 274, 275, 304
непрямая 306
при помощи подпрограмм 276
прямая 306
Интернет 28, 29, 542
NSF 544
интерпретатор 65
инфиксная запись 336
инфиксное вычисление 338
исключения 484
генерация исключений 484, 485
обработчики исключений 484
перехват исключений 486
распространение исключений 486
реализация 487
исполняемые объекты данных 272

использование псевдонимов 401
  трудности 403
история развития языков программирования 36

К

карта Холлерита 24
Кей А. 319
кембриджская польская запись 336
классификация 327
клиент-сервер 28, 526
ключевые слова 99
Кнут Д. 142, 159, 533
Ковальский Р. 368
когерентность кэш-памяти 528
код спагетти 46
коллизии 271
командный интерпретатор shell 57
командный процессор shell 34
комментарии 99
компилятор 74
компиляция 73
комплексные числа 212
компонент 238
компьютер 64
  CISC 67, 518
  MIMD 521
  RISC 67, 519
  адресация 67
  архитектура фон Неймана 70
  буферная память (кэш) 69
  виртуальный 65
  внешние файлы 66
  доступ к данным 68
  интерпретатор 65, 67
  компакт-диск 66
  компьютеры с массовым параллелизмом 522
  матричный переключатель 522
  мультипрограммирование 69
  мультипроцессор 71, 521
  омега-сеть 522
  оперативная память 65
  программно-аппаратный 72
  программно-моделируемый 64
  развитие 516
  регистры 65
  скорость обработки данных 69
  состояние 71
  универсальные ЭВМ 23
  управление последовательностью действий 67
  физический 64
  элементарные операции 67
  эмуляция 73
конвейер 519
конечный автомат 123
  вычислительные возможности 127
конкатенация 217
конкретизация 327
Консорциум WWW 561
константа 183
контекстно-зависимая грамматика 146
  свойства 146
контекстно-свободная грамматика 114
  свойства 146
контроль типов 195
  динамический 195
  статический 195
концептуальная целостность 38
концепции абстракций 325
  агрегация 327
  группирование 327
  декомпозиция 327
  индивидуализация 327
  классификация 327
  конкретизация 326
  обобщение 327
  создание экземпляров 327
критическая область 514
Кулмероз А. 368
куча 459
  управление памятью 464
кэш 66, 69

Л

лексема 106
ленивое вычисление 345
линейно-ограниченный автомат 154
логические значения 214
логическое программирование 332
локальная среда ссылок 409
локальные вычислительные сети 28
Лори Р. 549
Лукашевич В. 21, 335
Лэмпорт Л. 533

М

магазинный автомат 130
Маддукс Р. 363
Мак-Карти Дж. 22, 465
Маккуин Д. 169
макропроцессор 74
макрос 109
Маскитти Р. 293
массив
  сечение 570
массивы 252
  виртуальный начальный адрес 254
  индексация 254
  многомерные 252
  реализация 252
  сечения 255
матрица 252
матричный переключатель 522
метод наиболее подходящего 477
метод первого подходящего 476
методы 313
микрокомпьютер 25

микропрограмма 72
Миллз X. 158, 366
Милнер Р. 168
мини-компьютер 25
множества 268
битовые строки 269
операции 268
равенство 290
реализация 269
множественное наследование 316
модель спецификаций 158
модем 545
модули 102
монитор 514
Мошер Э. 549
МП-автомат 130
МТИ 465
мультипрограммирование 69
мультипроцессор 493, 521
мусор 244

Н

набор символов 97
APL 98
интернационализация 98
наследование 191, 311
абстрактные классы 318
множественное 316
основанное на делегировании 316
основанное на копировании 316
правило определения области видимости 311
производные классы 313
смешанное (mixin) 319
наследственная неоднозначность 120
Наур П. 20
недетерминированное выполнение 496
недетерминированный магазинный автомат (PDA) 131
независимое определение подпрограмм 101
Нейман Дж. 70
необязательные слова 99, 101
неоднозначность 96, 120, 152
условные операторы языка Ada 97
условный оператор 96
непосредственный дескриптор 329
неразрешимость 147
нетерминальный символ 116
НП-полнота 155
НФБ-грамматика 114, 142
LALR 132
LR(k) 133
выражения и присваивание 116
дополнительные способы записи 121
палиндромы 131
расширенная 121
рекурсивный спуск 133
сбалансированные последовательности круглых скобок 117
синтаксический анализ 115
синтаксический анализ строки 118

О

области применения языков
издательская деятельность 33
искусственный интеллект 32, 33
научные вычисления 31, 33
обработка деловой информации 31, 32
процессы 34
системная область 32, 33
область видимости 404
обобщение 327
обратная польская запись 336
обратная польская префиксная запись 382
общая среда 435
общая среда ссылок
реализация 437
общие
операции 194
подпрограммы 283
объект данных 181
атрибуты 182, 188
время жизни 182
значение 188
инициализация 204
реализация 192
связывания 183
способ представления 191
элементарный 187
объектно-ориентированный 27
объектно-ориентированное программирование 316
объектный язык 73
объекты данных
использование псевдонимов 401
эквивалентность 290
объявление 193
неявное 193
явное 193
омега-сеть 522
операторы 101
break 352
case 357
continue 353
составной оператор 356
do-while-do 362
goto 351
if 357
while 360
ввода 350
задачи 502
исключительные ситуации 362
недетерминированное выполнение 497
неопределенное число повторений 360
параллельное выполнение 495
повторение, основанное на данных 360
повторение, пока увеличивается счетчик 360
присваивания 349
простое повторение 358
реализация операторов цикла 361
реализация условных операторов 358
структурированные 101
структурное программирование 354

операторы (*продолжение*)
условные операторы 357
цикла 358
операторы (Prolog) 376
операции обработки ссылок 400
операционная модель 157
операционная система 80
операционная среда 23
операция 188
арность 190
как функция 189
сигнатура 189
ориентированные графы 267
ортогональность (свойство языка) 38
основанный на системе правил язык 45
откат 378
открытые объекты 315, 614
отладка 55
охраняемые команды 496, 508
if 498
оператор повторения 498
сторожевое условие 498
очереди 267

П

пакетная обработка данных 24
пакеты (Ada) 303
модификатор private 307
память
уплотнение 478
фрагментация 478
парадигмы языка 43
аппликативный язык 43
объектно-ориентированный язык 45
основанные на системе правил язык 45
параллельное выполнение 495
реализация 495
параллельное выполнение программы 493
параллельное программирование 493
языки 493
параметр 277
Паскаль Б. 135
первичная программа 363
перегрузка 55, 194, 283
передача параметров 415
in out параметр 423
входной параметр (in) 423
использование псевдонимов 430
метки операторов 435
по значению 420
по значению-константе 421
по значению-результату 421
по имени 419
по результату 422
по ссылке 420
подпрограммы как параметры 431
позиционное соответствие 418
примеры 425
реализация 424
связь с λ-выражениями 162
семантика 422
установка соответствия 418
передача параметров (*продолжение*)
фактический параметр 417
формальный параметр 417
явное имя 418
передача
по значению 420
по значению-константе 421
по значению-результату 421
по имени 419
по копии 420
по результату 422
по ссылке 420
перезапись элементов 372
переменная 183
чувствительная к регистру 183
переносимость программ 40
переходник (thunk) 419
перечисление 213
реализация 214
персональный компьютер 25
планирование подпрограмм 491
планируемые подпрограммы 491
побочный эффект 190, 346
повисшие ссылки 244, 469
повторно входимая подпрограмма 299
поддержка абстракций (свойство языка) 39
подпрограммы
активация 279, 387
ассоциация 399
запись активации 282
общие 283
перегрузка 283
пролог 424
простой вызов и возврат 386
процедура 277
реализация 278
рекурсия 393
сигнатура 277
совместно используемые данные 415
спецификация 277
управление последовательностью действий 384
формальные параметры 277
эпилог 425
подстановка 373
подтип 191
поисковые машины 548
полиморфизм 195, 328
полиномиальное время 155
порождение типа 310
портал 548
последовательный файл 223
постфиксная запись 336
вычисление 338
потоки управления 494
правила определения области видимости 55
правило активного вычисления 344
правило динамической области видимости 404
сохранение 410
уничтожение 410
правило ненулевой намотки 540
правило последней ассоциации 440

правило статической области видимости 405
блочная структура 407
правильная программа 363
предложение 376
преобразование типа 199
препроцессор 74
прерывания 505
префиксная запись 335
вычисление 337
приведение типа 199, 279
приоритеты 339
сочетательность 339
присваивание 201
пробелы 100
проблема останова 151
проблема узкого места фон Неймана 517
проверка корректности программы 169
программная имитация 75
программно-аппаратный компьютер 72
продукции 112
проект MAC 24
производные классы 313
реализация 315
пролог подпрограммы 424
простое имя 399
простота (свойство языка) 38
протокол 543
file 553
FTP 544
gopher 553
HTTP 546
mailto 553
news 553
SMTP 543
Telnet 543
стек TCP/IP 543
прототип функции 189
процессы 33
пустой указатель 220
путь доступа 244

Р

развертывание по столбцам 252
разделение времени 24
раздельная компиляция 53
вызывающая последовательность 53
Райс С. 549
рандеву 508
распределенная обработка данных 28
расширение 199
рациональные числа 211
регистр программных адресов 67
регулярная грамматика 113, 125
свойства 145
регулярные выражения 126
редактор см. AppBrowser
режим реального времени 27
резолюция 380
предложение 380
хорновы дизъюнкты 380
рекурсивная функция 395
рекурсивные подпрограммы 393
Ритчи Д. 58
Робинсон Дж. 380
Руссел Ф. 368

С

C++
производные классы 314
Саммет Дж. 16
самодокументируемая программа 94
самоизменение 190
сбор мусора 471
алгоритм маркировка-сбор 472
свободная переменная 432
своевременность 49
свойства языка
единообразие понятий 38
естественность 39
концептуальная целостность 38
неоднозначность 96
ортогональность 38
переносимость 40
поддержка абстракций 39
простота 38
среда программирования 40
стоимость использования 40
удобство верификации 39
удобство написания 95
удобство трансляции 95
удобство чтения 94
ясность 38
свойство Черча-Россера 162
семантика 42, 92, 156
семантические модели
аксиоматические 158
алгебраические типы данных 158
аппликативные 157
атрибутивные грамматики 159
модель спецификации 158
операционные 157
семафор 505
операция signal 505
операция wait 506
сентенциальная форма 118
сети 28, 525
сечения 255
реализация 255
сигнал 484, 504
сильно типизированный язык 198
символ 216
реализация 216
символы операций 94
синтаксис 41, 92
с фиксированными полями 100
со свободными полями 100
синтаксическая категория 116
синтаксическая схема 122
синтаксический анализ
детерминированный автомат 132
рекурсивный спуск 133

синхронизация 504
- прерывания 505
- рандеву 508
- семафоры 505
- сообщения 507

скалярные объекты данных 205
скобки 100
Скотт Д. 158
сложность 153
совместно используемые данные 415
создание экземпляров 327
сокрытие информации 273, 275
сообщение (Smalltalk) 320
сообщения 543
соответствие 50
сопоставление с образцом 219, 369
сопрограммы 489
- реализация 490

составная программа 364
составное имя 399
состояние компьютера 71
сохранение локальных переменных 410
- реализация 412

сохраняемость 185, 524
сочетательность 339
списки
- head 267
- tail 267
- деревья 267
- ориентированные графы 267
- очереди 267
- реализация 266
- свойств 267
- синтаксис 265
- стеки 267

список 265
список описаний 268
список свойств 618
среда
- встроенная компьютерная система 26
- интерактивная 24
- пакетная 23
- программирования 40, 53

глобальных ссылок 401
локальных ссылок 400
нелокальных ссылок 400
разработки 56
ссылок 400
- видимость 401
- глобальная 401
- локальная 400
- нелокальная 400
- реализация 411

ссылочный тип 220
стандартизация 47
- аномалия спецификации forward в языке Pascal 396
- интернационализация 51
- не рекомендуемая возможность 51
- своевременность 49
- согласительная 48
- соответствие 50
- устаревание 50
- устаревшая возможность 51

статическая область видимости 405
- важность 405
- реализация 443

статическая цепь 446
стек 267
стек постфиксного выполнения 536
стек-кактус 513
Стендфорский университет 136, 142
Стил Г. 465
стоимость использования (свойство языка) 40
страничная организация памяти 69
Страуструп Б. 21, 88
строка символов 217
- выбор подстроки 218
- неограниченной длины 217
- переменной длины 217
- реализация 219
- синтаксис 217
- сопоставление с образцом 219
- фиксированной длины 217

структурная теорема 366
структурная эквивалентность 289
структурное программирование 354, 361
- первичные программы 363, 366

структуры данных 238
- атрибуты 238
- записи 257
- контроль типов 245
- операции 239
- последовательное представление 241
- представление в памяти 241
- реализация 242
- связанное представление 241
- управление ресурсами памяти 244

Стрэчи К. 158
схема упорядочения 52
сценарий shell 127
счетчик команд 67
счетчики ссылок 471
Сьюсманн Д. 465

Т

таблица локальной среды 411
тезис Черча 150
Тейтелман В. 465
текстовый файл 224
терминальные символы 116
тип
- вывод 198
- определение 285, 291
- преобразование 199
- реализация 287, 292
- структурная эквивалентность 288
- эквивалентность имен 288

тип данных 186
- примитивный 186
- реализация 187
- синтаксическое представление 187
- спецификация 187

типы объединений 263
Томпсон К. 58
точка останова 56

транслятор 65
генерация кода 111
загрузка 112
лексический анализ 106
макрообработка 109
обнаружение ошибок 109
оптимизация 110
поддержка таблицы символов 108
проходы 106
семантический анализ 107
синтаксический анализ 107
трансляция 73
интерпретация 530
компиляция 531
семантическое описание 531, 548
трассировка 55
трехслойная архитектура клиент-сервер 527

У

удаление локальных переменных
реализация 413
удобство чтения программ
(свойство языка) 94
удобство написания программ (свойство языка) 95
указатели
мусор 244
повисшие ссылки 244, 469
указатель 220
this 596
операция разыменования 220
операция создания 220
реализация 222
статической цепи 446
укороченное вычисление 348
укороченное вычисление
булевых выражений 347
унарная операция 190
унаследованный атрибут 159
универсальные ЭВМ 23
универсальный компьютер 29
универсальный язык программирования 148
унификация 373, 374
унифицированное вычисление
выражений 344
уничтожение локальных переменных 410
Уолл Л. 128
Уорнок Д. 536
уплотнение 463
управление доступом 614
управление заданиями 57
управление памятью 459
восстановление 468
куча 464
метод наиболее подходящего 477
метод первого подходящего 476
статическое 463
стековая реализация 391
счетчики ссылок 471
уплотнение 478
фазы 463
фрагментация 478
элементы переменного размера 474
управление последовательностью
действий 67, 332
выражения 333
операторы 348
подпрограммы 384
устаревание 50
утверждение 56, 488

Ф

файл 223
буфер 226
индексно-последовательный 223, 229
интерактивный 227
последовательный 223
прямого доступа 223, 228
реализация 226
текстовый 224, 227
фактический параметр 417
Флойд Р. 143
форма Бэкуса-Наура 21
формальный параметр 417
фрагментация 478
фрагментированная память 269
функциональная семантика 158
функциональный способ конструирования
программы 274

Х

хвост списка 267
Холлерит Г. 24
Хомский Н. 21, 114
Хоор Т. 143, 158
хорновы дизъюнкты 380
Хупер Г. 19, 22
хэш-кодирование 269
алгоритм 271
коллизии 271
хэш-таблица 270

Ц

целые числа 205
перечисления 213
поддиапазон 207
представление в памяти 206
центральная таблица
среды ссылок 441

Ч

Черч А. 150
числа с плавающей точкой 208
реализация 209
формат IEEE 210
числа с фиксированной точкой 211
реализация 211

Ш

шаблон 310
Шарпф Н. 549

Шварц Ж. 21
шестнадцатеричные цифры 558
шрифты
  Helvetica 650
  Times Roman 650
  пункты 538

Э

Эдинбургский университет 368
эквивалентность 203
  имен 288
  типов 287
экземпляр 373
электронная коммерция 30, 33
элементарный объект данных 187
эмуляция 73

Я

явно определяемая общая среда 435
язык
  формальное определение 115
  формальные свойства 143
язык сценариев 57
языки описания страниц 535
языки программирования
  4GLs 32
  Ada 499, 564
  ALGOL 20
  ALGOL 68 21
  ALGOL-W 21
  BCPL 23
  C 58, 581
  C++ 293, 593
  COBOL 22
  COMIT 23
языки программирования (*продолжение*)
  CORC 113
  CPL 23
  FLOWMATIC 22
  Forth 342
  FORTRAN 19
  FORTRAN 66 20
  FORTRAN 77 20
  FORTRAN 90 20
  FORTRAN II 20
  FORTRAN IV 20
  IAL 20
  IPL 22
  Java 558
  JOVIAL 21
  LaTeX 533
  LISP 22, 465, 616
  ML 624
  MPPL 22
  NPL 21
  Pascal 21, 135, 635
  Perl 34, 127, 645
  PL/I 22, 113
  Postscript 33, 536, 649
  Prolog 23, 368, 653
  Simula-67 21
  Smalltalk 319, 659
  SNOBOL 23
  SNOBOL4 371
  TCL 34
  TeX 33
  интерпретируемые 77
  компилируемые 77
  C+- 293
языки программирования с сохраняемыми данными 524
Якобини В. 366
ясность (свойство языка) 38

Т. ПРАТТ, М. ЗЕЛКОВИЦ

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

РАЗРАБОТКА И РЕАЛИЗАЦИЯ

4-Е ИЗДАНИЕ

КНИГИ, КОТОРЫЕ НЕ СТАРЕЮТ!

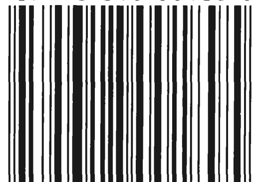
КЛАССИКА COMPUTER SCIENCE

В книге известных американских специалистов **Теренса Пратта** и **Марвина Зелковица** рассматриваются общие концепции разработки языков программирования, а также основы формальных грамматик и конечных автоматов — математических моделей, используемых для их определения и реализации.

Трудно представить себе программиста, не читавшего этой книги. Ее прототип появился еще четверть века назад, и на нем воспитывалось целое поколение первопроходцев программирования. Языки программирования рождались, старели и умирали, и содержание книги изменялось, отражая эти процессы.

У вас в руках — уже четвертое издание, в котором помимо важнейших языков конца XX века рассматриваются и новые, появление которых обусловлено бурным развитием программирования для бизнеса и World Wide Web.

ISBN 5-318-00189-0



Посетите наш web-магазин: www.piter.com

PH PTR