С. З. СВЕРДЛОВ

ЯЗЫКИ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

Издание второе, исправленное

• САНКТ-ПЕТЕРБУРГ •
• МОСКВА • КРАСНОДАР •
• 2019 •

УДК 004.43
ББК 32.973.26-018.1я73
С 24

Свердлов С. З.

С 24 Языки программирования и методы трансляции: Учебное пособие. — 2-е изд., испр. — СПб.: Издательство «Лань», 2019. — 564 с.: ил. — (Учебники для вузов. Специальная литература).

ISBN 978-5-8114-3457-2

В книге рассматриваются вопросы сравнительного анализа языков программирования и конструирования компиляторов.

Первая часть книги содержит обзор языков высокого уровня и связанных с их эволюцией технологий структурного, модульного и объектно-ориентированного программирования. Проводится сравнительный анализ языков, в том числе на основе объективных критериев, даются экспертные оценки.

Вторая тема книги — конструирование компиляторов. Обсуждаются все элементы транслятора и этапы реализации языка от спецификации до формирования машинного кода.

Книга адресуется студентам вузов, специализирующимся по компьютерным технологиям, программистам-практикам и всем, кто интересуется программированием.

УДК 004.43
ББК 32.973.26-018.1я73

Рецензенты:

В. О. САФОНОВ — доктор технических наук, профессор Санкт-Петербургского государственного университета;
В. А. СУХОМЛИН — доктор технических наук, профессор, зав. лабораторией открытых информационных технологий факультета вычислительной математики и кибернетики Московского государственного университета им. М. В. Ломоносова;
А. Н. ТЕРЕХОВ — доктор физико-математических наук, профессор, зав. кафедрой системного программирования Санкт-Петербургского государственного университета.

Обложка
Е. А. ВЛАСОВА

© Издательство «Лань», 2019
© С. З. Свердлов, 2019
© Издательство «Лань», художественное оформление, 2019

Оглавление

ПРЕДИСЛОВИЕ К ПЕРВОМУ ИЗДАНИЮ 9
ОТ АВТОРА 11
ГЛАВА 1. ЯЗЫКИ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ 12
Язык и его реализация 12
Компилятор, интерпретатор, конвертор 13
Метаязыки 18
Генеалогия языков программирования 19
Первое поколение языков 20
Фортран 20
Алгол-60 30
Кобол 40
Две попытки объять необъятное 42
ПЛ/1 42
Алгол-68 47
Интерактивное программирование для всех 55
Бейсик 56
Структурное программирование 69
Основы структурного программирования 73
Паскаль 87
Язык Си 96
Модульность, надежность, абстракция 106
Ада 106
Модула-2 112
Абстрактные типы данных 119
Объектно-ориентированное программирование 121
Язык программирования Си++ 132
Язык программирования Оберон 139
Язык программирования Ява 152
Язык программирования Си# 166
Примеры использования объектной технологии 171
Языки-концепции 202
Форт 203
Лисп 206
Пролог 207

Смолток 207
Языки Интернета 209
HTML 209
Ява и аплеты 210
Скриптовые языки 212
Языки CGI-программирования 213
Языки активных серверных страниц 217
Языки Интернета: повторение пройденного 219
Какой язык лучше. Сравнительная оценка языков программирования 221
Арифметика синтаксиса 223
Важнейшие языки 230
ГЛАВА 2. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ТРАНСЛЯЦИИ 231
Формальные языки и грамматики 231
Основные термины и определения 231
Примеры языков 233
Порождающие грамматики (грамматики Н. Хомского) 234
Еще несколько определений 238
Дерево вывода 240
Задача разбора 241
Для чего надо решать задачу разбора 242
Домино Де Ремера 242
Разновидности алгоритмов разбора 244
Эквивалентность и однозначность грамматик 245
Иерархия грамматик Н. Хомского 247
Автоматные грамматики и языки 250
Граф автоматной грамматики 250
Конечные автоматы 252
Преобразование недетерминированного конечного автомата (НКА) в детерминированный конечный автомат (ДКА) 253
Таблица переходов детерминированного конечного автомата 256
Программная реализация автоматного распознавателя 257
Дерево разбора в автоматной грамматике 258
Пример автоматного языка 259
Синтаксические диаграммы автоматного языка 262
Регулярные выражения и регулярные множества 264

Эквивалентность регулярных выражений и автоматных грамматик 266
Для чего нужны регулярные выражения 267
Регулярные выражения как языки 268
Расширенная нотация для регулярных выражений 268
КОНТЕКСТНО-СВОБОДНЫЕ (КС) ГРАММАТИКИ И ЯЗЫКИ 269
Однозначность КС-грамматики 269
Алгоритмы распознавания КС-языков 270
Распознающий автомат для КС-языков 271
Самовложение в КС-грамматиках 271
Синтаксические диаграммы КС-языков 272
Определение языка с помощью синтаксических диаграмм 275
Синтаксический анализ КС-языков методом рекурсивного спуска 278
Требование детерминированного распознавания 286
LL-грамматики 287
Левая и правая рекурсия 288
Синтаксический анализ арифметических выражений 288
Включение действий в синтаксис 296
Обработка ошибок при трансляции 306
Табличный LL(1)-анализатор 310
Рекурсивный спуск и табличный анализатор 319
ТРАНСЛЯЦИЯ ВЫРАЖЕНИЙ 320
Польская запись 320
Алгоритм вычисления выражений в обратной польской записи 321
Перевод выражений в обратную польскую запись 324
Интерпретация выражений 326
Семантическое дерево выражения 327
Упражнения для самостоятельной работы 339
ГЛАВА 3. ТРАНСЛЯЦИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ 346
ОПИСАНИЕ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ 346
Метаязыки 347
БНФ 347
Синтаксические диаграммы 348
Расширенная форма Бэкуса — Наура (РБНФ) 348
Описания синтаксиса языков семейства Си 349

Описания синтаксиса языка Ада 350
Язык программирования «О»................................. 351
Краткая характеристика языка «О» 351
Синтаксис «О» 352
Пример программы на «О»................................. 354
Структура компилятора 355
Многопроходные и однопроходные трансляторы 357
Компилятор языка «О»................................. 359
Вспомогательные модули компилятора 361
Лексический анализатор (сканер) 363
Виды и значения лексем 365
Лексический анализатор языка «О»................................. 366
Синтаксический анализатор................................. 383
Контекстный анализ 387
Таблица имен................................. 388
Контекстный анализ модуля 397
Трансляция списка импорта................................. 400
Трансляция описаний................................. 402
Контекстный анализ выражений 405
Контекстный анализ операторов................................. 409
Генерация кода................................. 412
Виртуальная машина 412
Архитектура виртуальной машины................................. 413
Программирование в коде виртуальной машины................................. 420
Реализация виртуальной машины................................. 425
Генератор кода 431
Распределение памяти................................. 433
Генерация кода для выражений 435
Генерация кода для операторов................................. 448
Завершение генерации 458
Назначение адресов переменным................................. 459
Трансляция процедур 462
Расширенный набор команд виртуальной машины 463
Процедуры без параметров и локальных переменных 464
Процедуры с параметрами-значениями без локальных переменных 466
Процедуры с параметрами-значениями и локальными переменными 470

Простейшая оптимизация кода 472
Процедуры-функции с параметрами-значениями и локальными переменными 472
Трансляция оператора RETURN 475
Особенность трансляции параметров-переменных 475
Пример программы на языке «О с процедурами» 477
Конструкция простого ассемблера 482
Язык ассемблера виртуальной машины 482
Реализация ассемблера 488
Автоматизация построения и мобильность трансляторов 496
Автоматический анализ и преобразование грамматик 496
Автоматическое построение компилятора и его частей 497
Использование языков высокого уровня 503
Самокомпилятор. Раскрутка 506
Примеры раскрутки 510
Унификация промежуточного представления 511
ПРИЛОЖЕНИЕ. ЯЗЫК ПРОГРАММИРОВАНИЯ ОБЕРОН-2 517
От переводчика 517
1. Введение 519
2. Синтаксис 520
3. Словарь и представление 520
4. Объявления и области действия 522
5. Объявления констант 523
6. Объявления типов 524
6.1. Основные типы 524
6.2. Тип массив 525
6.3. Тип запись 526
6.4. Тип указатель 527
6.5. Процедурные типы 527
7. Объявления переменных 527
8. Выражения 528
8.1. Операнды 528
8.2. Операции 529
9. Операторы 532
9.1. Присваивания 533
9.2. Вызовы процедур 533

9.3. Последовательность операторов 534
9.4. Операторы IF 534
9.5. Операторы CASE 535
9.6. Операторы WHILE 535
9.7. Операторы REPEAT 536
9.8. Операторы FOR 536
9.9. Операторы LOOP 537
9.10. Операторы возврата и выхода 537
9.11. Операторы WITH 538
10. Объявления процедур 538
10.1. Формальные параметры 539
10.2. Процедуры, связанные с типом 541
10.3. Стандартные процедуры 542
11. Модули 544
Приложение A: Определение терминов 546
Целые типы 546
Вещественные типы 546
Числовые типы 546
Одинаковые типы 546
Равные типы 547
Поглощение типов 547
Расширение типов (базовый тип) 547
Совместимость по присваиванию 547
Совместимость массивов 548
Совместимость выражений 548
Совпадение списков формальных параметров 549
Приложение B: Синтаксис Оберона-2 549
Приложение C: Модуль SYSTEM 551
Приложение D: Среда Оберон 553
D1. Команды 553
D2. Динамическая загрузка модулей 555
D3. Сбор мусора 555
D4. Смотритель 556
D5. Структуры данных времени выполнения 556
ЛИТЕРАТУРА 558

Предисловие к первому изданию

Программист не может не интересоваться языками программирования. Даже если в своей работе или учебе вы пользуетесь каким-то одним языком, иметь представление о других, их свойствах, областях применения, истории и современном состоянии полезно и важно.

Первая часть книги содержит обзор языков высокого уровня и связанных с их эволюцией технологий структурного, модульного и объектно-ориентированного программирования. Проводится сравнительный анализ языков, в том числе на основе объективных критериев, даются экспертные оценки. Подробно обсуждаются Ява и Си#. Отдельный раздел посвящен языкам Интернета. Изложение сопровождается большим числом примеров программ. Знакомство с этим обзором поможет составить предметное представление о свойствах основных языков, обоснованно подходить к их выбору.

Реальное применение языка программирования невозможно без соответствующей системы программирования, основу которой составляет транслятор. Конструирование трансляторов — вторая тема книги. Транслятор — весьма непросто устроенная программа, разработка которой невозможна без знакомства с элементами теории формальных языков и соответствующей техникой программирования.

Создание компилятора или интерпретатора — увлекательное дело, дающее возможность совершенствовать свои программистские навыки, глубже понять устройство самих языков. Конечно, не каждому программисту доводится участвовать в разработке транслятора и уж тем более в создании языка программирования. Но знание методов программирования трансляторов оказывается полезным в самых разных ситуациях: от решения олимпиадных задач до разработки и реализации входных языков прикладных систем.

В книге обсуждаются все элементы транслятора и этапы реализации языка от спецификации до формирования машинного кода. Приводится полный исходный текст компилятора на нескольких языках программирования. Рассматривается конструкция простого двухпроходного ассемблера. Обсуждаются возможности автоматизации построения трансляторов и способы повышения их мобильности.

Особую роль в изложении играет язык Оберон (и его расширение Оберон-2) — один из самых совершенных и современных языков программирования, разработанный Н. Виртом — автором Паскаля и

Модулы-2. Оберон необычайно прост и в то же время содержит все необходимые средства структурного, объектно-ориентированного и модульно-компонентного программирования. В книге публикуется спецификация Оберон-2. Трансляция языков программирования рассматривается на примере разработки компилятора для подмножества Оберона. Надеюсь, что читатели с удовольствием познакомятся с этим языком, даже если не обязательно в дальнейшем будут использовать его.

Книга написана по материалам одноименного курса, который в течение ряда лет читается студентам факультета прикладной математики и компьютерных технологий Вологодского педагогического университета. В работе над курсом и книгой я руководствовался несколькими основными принципами: изложение должно быть понятным, конструктивным и интересным. В какой мере это удалось — судить вам. Некоторые суждения, высказанные в книге, заведомо субъективны. Вы можете с ними не согласиться, но, надеюсь, признаете, что они аргументированы.

Книга адресуется студентам вузов, специализирующимся по компьютерным технологиям, программистам-практикам и всем, кто интересуется программированием. Предполагается, что читатель имеет начальные навыки программирования и обладает математической подготовкой в пределах программы средней школы.

Мне доставляет удовольствие поблагодарить тех, кто помог советом и делом в работе над книгой. Хочу выразить признательность блистательному создателю Паскаля, Модулы и Оберона, профессору Н. Вирту, который ответил на мои письма и вместе с профессором Х. Мёссенбёком дал согласие на опубликование русского перевода спецификации Оберона-2. На протяжении всей работы я имел удовольствие дружеского общения с Е. А. Зуевым, который оказал неоценимую поддержку и поделился множеством интересных соображений. Я благодарен рецензентам — профессору В. А. Сухомлину, профессору В. О. Сафонову, высказавшему множество ценных замечаний, и профессору А. Н. Терехову, который также был моим консультантом по Алголу-68.

В течение ряда лет студенты факультета прикладной математики и компьютерных технологий ВГПУ, проходившие курс, сделали немало предложений и замечаний, способствовавших его улучшению. Большое влияние на содержание книги и мое понимание проблемати-

ки языков программирования и методов трансляции оказал Ф. В. Меньшиков. Он, а также Я. А. Музыкантов и В. С. Губа стали первыми читателями рукописи. Их поправки были очень ценны. Благодарю за консультации И. Р. Агамирзяна.

От автора

Во втором издании книги исправлены замеченные ошибки и опечатки. Откорректированы некоторые оценки и выводы с учетом тех изменений, которые произошли со времени выхода первого издания, сокращен материал, потерявший актуальность.

Из книги исключены приложения, содержащие текст компилятора, имевшиеся в первом издании. Этот материал можно получить на сайте автора:
http://telegra.ph/YAzyki-programmirovaniya-i-metody-translyacii-03-02.

Глава 1. Языки программирования высокого уровня

С середины 50-х годов XX века создаются и развиваются *языки программирования высокого уровня*. Не пытаясь дать определение этому термину, скажем лишь, что речь идет о языках, запись программы на которых приближена к обычной математической, удобна для программиста и достаточно далека по форме от программ в машинном коде, которые могут непосредственно исполняться компьютером. Язык высокого уровня позволяет оперировать такими понятиями как переменная, объект, ссылка, тип, класс, процедура, метод, избавляя программиста от необходимости манипулировать такими низкоуровневыми, свойственными машинному языку вещами, как регистр, адрес, ячейка, переход. Подавляющее число программ создается с использованием языков высокого уровня.

Язык и его реализация

Условимся о некоторых терминах. Рассматривая каждый язык, мы будем говорить о его разработчиках и дате создания. Но что такое дата создания языка? Появление языка программирования — это не одномоментный процесс. Да и недостаточно придумать язык (на что тоже требуется время), надо еще сделать так, чтобы его можно было реально применять — писать программы на этом языке и исполнять их на компьютере. Язык должен быть *реализован*.

Реализацией языка программирования называют создание комплекса программ, обеспечивающих работу на этом языке. Такой набор программ называется *системой программирования*.

Основу каждой системы программирования составляет *транслятор*. Это программа, переводящая текст на языке программирования в форму, пригодную для исполнения (на другой язык). Такой формой обычно являются машинные команды, которые могут непосредственно исполняться компьютером. Совокупность машинных команд данного компьютера (процессора) образует его *систему команд* или *машинный язык*. Программу, которую обрабатывает транслятор, называют *исходной программой*, а язык, на котором записывается исходная программа — *входным языком* этого транслятора.

Компилятор, интерпретатор, конвертор

Различают несколько видов трансляторов: компиляторы, интерпретаторы, конверторы (рис. 1.1).

Компилятор, обрабатывая исходную программу, создает эквивалентную программу на машинном языке, которая называется также *объектной программой*, или *объектным кодом*. Объектный код, как правило, записывается в файл, но не обязательно представляет собой готовую к исполнению программу. Для программ, состоящих из многих модулей, может образовываться много объектных файлов. Объектные файлы объединяются в исполняемый модуль с помощью специальной *программы-компоновщика*, которая входит в состав системы программирования. Возможен также вариант, когда модули не объединяются заранее в единую программу, а загружаются в память при выполнении программы по мере необходимости.

Интерпретатор, распознавая, как и компилятор, исходную программу, не формирует машинный код в явном виде. Для каждой операции, которая может потребоваться при исполнении исходной программы, в программе-интерпретаторе заранее заготовлена машинная команда или команды. «Узнав» очередную операцию в исходной программе, интерпретатор выполняет соответствующие команды, потом — следующие, и так всю программу. Интерпретатор — это переводчик и исполнитель исходной программы.

Различие между интерпретаторами и компиляторами можно проиллюстрировать такой аналогией. Чтобы пользоваться каким-либо англоязычным документом, мы можем поступить по-разному. Можно один раз перевести документ на русский, записать этот перевод и потом пользоваться им сколько угодно раз. Это ситуация, аналогичная компиляции. Заметьте, что после выполнения перевода переводчик (компилятор) больше не нужен.

Интерпретация же подобна чтению и переводу «с листа». Перевод не записывают, но всякий раз, когда нужно воспользоваться документом, его переводят заново, каждый раз при этом используя переводчика[1].

Нетрудно понять преимущества и недостатки компиляторов и интерпретаторов.

---

[1] Одно из основных значений английского interpreter — устный (синхронный) переводчик.

Рис. 1.1. Схема работы компилятора и интерпретатора

Компилятор обеспечивает получение быстрой программы на машинном языке, время работы которой намного меньше времени, которое будет затрачено на исполнение той же программы интерпретатором. Однако компиляция, являясь отдельным этапом обработки программы, может потребовать заметного времени, снижая оперативность работы. Откомпилированная программа представляет собой самостоятельный продукт, для использования которого компилятор не требуется. Программа в машинном коде, полученная компиляцией, лучше защищена от внесения несанкционированных искажений, раскрытия лежащих в ее основе алгоритмов.

Интерпретатор исполняет программу медленно, поскольку должен распознавать конструкции исходной программы каждый раз, когда они исполняются. Если какие-то действия расположены в цикле, то распознавание одних и тех же конструкций происходит многократно. Зато интерпретатор может начать исполнение программы сразу же, не затрачивая времени на компиляцию, — получается оперативней. Интерпретатор, как правило, проще компилятора, но его присутствие

требуется при каждом запуске программы. Поскольку интерпретатор исполняет программу по ее исходному тексту, программа оказывается незащищенной от постороннего вмешательства.

Но даже при использовании компилятора получающаяся машинная программа работает, как правило, медленнее и занимает в памяти больше места, чем такая же программа, написанная вручную на машинном языке или языке ассемблера квалифицированным программистом. Компилятор использует набор шаблонных приемов для преобразования программы в машинный код, а программист может действовать нешаблонно, отыскивая оптимальные решения. Разработчики тратят немало усилий, стремясь улучшить качество машинного кода, порождаемого компиляторами.

В действительности различие между интерпретаторами и компиляторами может быть не столь явным. Некоторые интерпретаторы выполняют предварительную трансляцию исходной программы в промежуточную форму, удобную для последующей интерпретации. Некоторые компиляторы могут не создавать файла объектного кода, а компилировать программу в память, не записывая машинный код в файл, и тут же ее запускать. Такое поведение вполне соответствует работе интерпретатора.

Существуют трансляторы, переводящие программу не в машинный код, а на другой язык программирования. Такие трансляторы иногда называют *конверторами*. Например, в качестве первого шага при реализации нового языка часто разрабатывают конвертор этого языка в язык Си. Дело в том, что Си — один из самых распространенных и хорошо стандартизованных языков. Обычно ориентируются на версию ANSI Си — стандарт языка, принятый Американским Национальным Институтом Стандартов (American National Standards Institute, ANSI). Компиляторы ANSI Си есть практически в любой системе.

Кросскомпиляторы (cross-compilers) генерируют код для машины, отличной от той, на которой они работают.

Пошаговые компиляторы (incremental compilers) воспринимают исходный текст программы в виде последовательности задаваемых пользователем *шагов* (шагом может быть описание, группа описаний, оператор, группа операторов, заголовок процедуры и др.); допускается ввод, модификация и компиляция программы по шагам, а также отладка программ в терминах шагов.

Динамические компиляторы (Just-in-Time — JIT compilers), получившие в последнее время широкое распространение, транслируют промежуточное представление программы в объектный код во время исполнения программы.

Двоичные компиляторы (binary compilers) переводят объектный код одной машины (платформы) в объектный код другой. Такая разновидность компиляторов используется при разработке новых аппаратных архитектур, для переноса больших системных и прикладных программ на новые платформы, в том числе — операционных систем, графических библиотек и др.

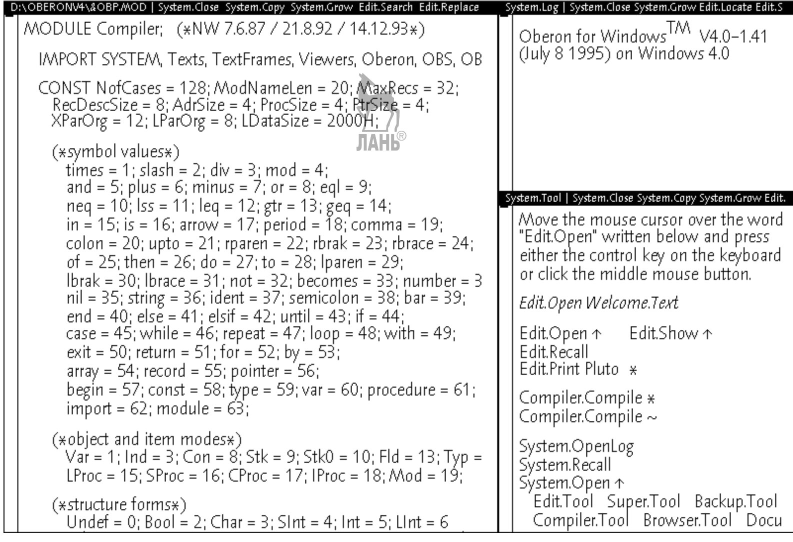


Рис. 1.2. Компилятор языка Оберон, написанный Никлаусом Виртом, с его «автографом» (NW). Фрагмент исходного текста основного модуля показан в левом окне Оберон-системы, частью которой является компилятор. Компилятор написан на языке Оберон и может компилировать сам себя

Транслятор — это большая и сложная программа. Размер программы-транслятора составляет от нескольких тысяч до сотен тысяч строк исходного кода. Вместе с тем разработка транслятора для не слишком сложного языка — задача вполне посильная для одного человека или

небольшого коллектива. Методы разработки трансляторов и будут рассмотрены в этой книге.

Транслятор связан в общем случае с тремя языками. Во-первых, это входной язык, с которого выполняется перевод. Во-вторых, целевой (объектный) язык, на который выполняется перевод. И, наконец, третий язык — это язык, на котором написан сам транслятор, — *инструментальный язык*. Трансляторы удобно изображать в виде Т-диаграмм[2] (рис. 1.3), которые предложил Х. Брэтман (H. Bratman) в 1961 году. Слева на такой диаграмме записывается исходный язык, справа — объектный, снизу — инструментальный.

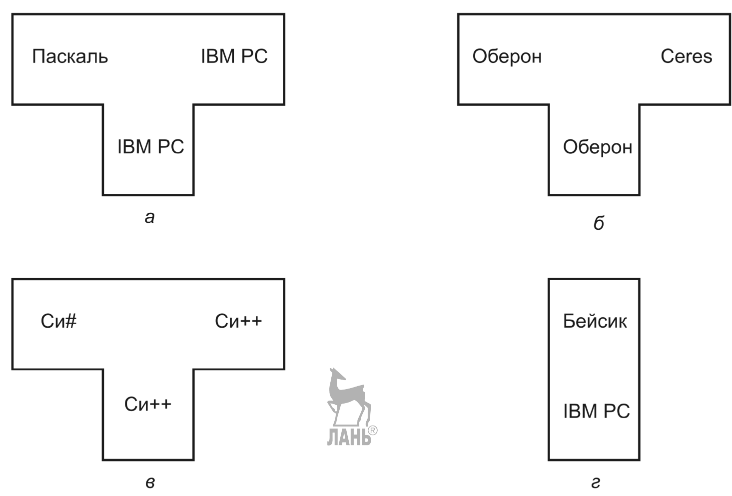


Рис. 1.3. Диаграммы трансляторов

Первые трансляторы, появившиеся в 1950-е годы, программировались на машинном языке или на языке низкого уровня — языке ассемблера (автокоде, как принято было говорить в то время). Сейчас для разработки трансляторов используются языки высокого уровня.

Изображая Т-диаграммы, мы можем не знать, на каком языке написан транслятор. В этом случае, а также когда недоступен исходный

---

[2] Кроме того, что диаграмма имеет форму буквы «Т», примечательно, что с этой буквы начинается и само слово «транслятор», как русское, так и английское.

текст транслятора (т. е. мы располагаем только его исполняемой откомпилированной версией), на Т-диаграмме в качестве инструментального записывают машинный язык или обозначение того компьютера, на котором этот транслятор работает. Можно считать, что на рисунке 1.3*а* изображен компилятор Turbo Pascal, Delphi или Free Pascal, транслирующий с языка Паскаль в машинный код IBM PC-совместимого компьютера и существующий в виде программы в машинном коде IBM PC. На рисунке 1.3*б* показана Т-диаграмма компилятора с языка Оберон (см. рис. 1.2), транслирующего в машинный код компьютера Ceres и написанного на языке Оберон. Рисунок 1.3*в* соответствует конвертору. Си# — язык программирования, созданный в корпорации Microsoft. Первая реализация Си# вполне могла быть выполнена по такой схеме. В дальнейшем мы еще обратимся к Т-диаграммам и узнаем, как они могут сочетаться подобно костям домино, иллюстрируя процесс получения одних трансляторов с помощью других.

Интерпретатор, кстати, можно изобразить в виде I-диаграммы (Interpreter — интерпретатор). Сверху записываем название входного, а внизу — инструментального языка. Пример диаграммы для интерпретатора Бейсика, работающего на IBM PC, можно видеть на рисунке 1.3*г*.

Метаязыки

Говоря о Т-диаграммах, мы забыли еще об одном, четвертом языке, который неизбежно присутствует в разговоре о трансляторах и языках программирования. Это язык, например, русский, на котором мы ведем сам разговор. Язык, используемый для описания других языков, называется *метаязыком.* В дальнейшем мы рассмотрим формализованные метаязыки, а пока в этой роли будем применять родной, естественный язык.

Понятно, что бессмысленно обсуждать сколько-нибудь содержательно сразу многие языки программирования, если не знаешь и одного, не имеешь хотя бы небольшого опыта программирования. Я рассчитываю, что у вас такой опыт есть. Я даже предполагаю, что вы, скорее всего, знакомы с языком Паскаль или, может быть, Си. Эти языки в нашем обзоре тоже будут в известном смысле играть роль метаязыков. Рассматривая другие языки, мы будем сравнивать имеющиеся в них средства с теми, что есть в Паскале или Си.

Генеалогия языков программирования

На рисунке 1.4 показано генеалогическое дерево языков высокого уровня. Влияние языков друг на друга несомненно. Это влияние обусловлено и тем, что некоторые специалисты участвовали в создании нескольких языков, и тем, что новые идеи, появляясь в одном языке, затем поддерживались и развивались в других. Существуют целые семейства языков. Пожалуй, самой многочисленной является линия алголоподобных языков, ведущая начало от Алгола-60, появившегося в 1960 году, сейчас уже не используемого, но оказавшего влияние на все последующее развитие. Явная, очевидная связь изображена на рисунке сплошной стрелкой, косвенное влияние — пунктирной.

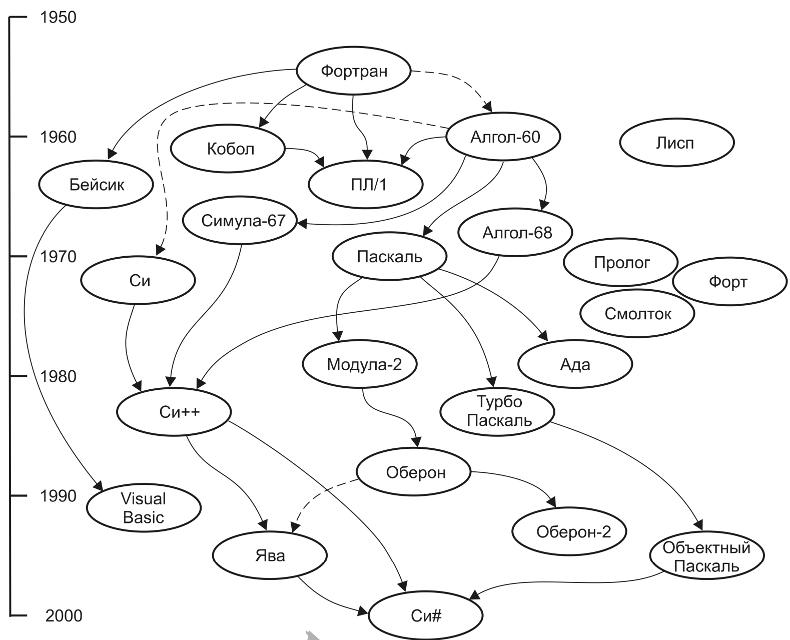


Рис. 1.4. Генеалогическое дерево языков программирования высокого уровня

Бурный рост числа языков программирования происходил в 1960-е годы. В языках программирования, появившихся в 70-е годы, нашла

воплощение идея структурного программирования. Распространение технологии объектно-ориентированного программирования и появление интернета дало толчок разработке новых языков в 80-е и 90-е годы. После 2000 года ситуация стабилизируется. Новых получивших широкое распространение универсальных языков программирования не появляется. И в 2016 году языки Ява, Си# и Си++ прочно занимают эту нишу.

Первое поколение языков

С конца 1940-х годов, то есть с момента создания первых вычислительных машин, предпринимались усилия по упрощению и автоматизации кодирования программ. Использовались библиотеки подпрограмм с автоматической компоновкой, создавались различные системы символического кодирования. Уже тогда в обиход были введены термины ассемблер (Морис Уилкс) и компилятор (Грейс Хоппер). Эти слова, кстати, в английском языке означают по сути одно и то же: сборщик, собиратель.

Прообразом современных языков высокого уровня является Планкалкюль — язык, созданный в 1945 году талантливым немецким инженером Конрадом Цузе (Konrad Zuse). Однако Планкалкюль так и не был реализован. Часто в связи с разговором о первых языках программирования упоминают ряд проектов, осуществленных Грейс Хоппер (Grace Hopper) в 1950-е годы.

Но первыми по-настоящему распространенными языками стали Фортран, Алгол и Кобол.

Фортран

Фортран (Fortran) считается первым языком программирования высокого уровня. Язык разработан в компании IBM. Работы по созданию языка и компилятора начались в 1954 году. Проект был инициирован сотрудником IBM Джоном Бэкусом, который обратился к руководству с предложением реализовать язык, облегчающий программирование выпущенного незадолго до этого компьютера IBM-704. Была создана группа из десяти человек под руководством Бэкуса. Компилятор окончательно отладили в 1957 году, и он стал распространяться среди пользователей IBM-704.

Несмотря на первоначальные опасения, что компилятор не сможет обеспечить качество машинного кода, сопоставимое с качеством руч-

ного кодирования, проект оказался очень успешным. Уже через год большинство программ для компьютеров IBM-704 записывалось на Фортране. Достаточной эффективности получаемой машинной программы удалось добиться благодаря тому, что язык был довольно простым и не содержал конструкций, порождающих неэффективный код.

Название языка происходит от сокращения слов FORmula TRANslation — перевод формул. Запись формул в привычном виде была одной из важных возможностей языка. Фортран был ориентирован на программирование вычислительных задач, в нем отсутствовали средства обработки нечисловой информации.

Уже к началу 1960-х годов Фортран был реализован на многих типах машин. Появились и стали популярными его новые версии Фортран II, Фортран IV. Важным моментом в судьбе Фортрана стала его стандартизация в 1966 году Американским Национальным Институтом Стандартов (ANSI). Появление ANSI-стандарта еще больше способствовало распространению языка. Именно в этой версии, практически совпадавшей с Фортраном IV, язык в течение 1960-х и 1970-х годов оставался самым распространенным языком программирования в мире.

В нашей стране пик популярности Фортрана пришелся на 1970-е годы и начало 1980-х. Тогда в СССР произошел переход от выпуска машин оригинальных архитектур к производству ЕС ЭВМ, которые повторяли серию компьютеров IBM/360, а затем IBM/370. Одним из самых популярных компиляторов на этих компьютерах был Фортран IV. Впрочем, Фортран применялся не только на ЕС ЭВМ с американским компилятором. Существовали и оригинальные отечественные реализации Фортрана для машин «Минск-22», «Минск-32», «БЭСМ-6» и других.

Основные черты языка Фортран

Здесь я привожу перечень основных свойств Фортрана IV — версии, распространенной в период расцвета этого языка.

Запись программы

В отличие от современных языков, допускающих свободный формат записи программы, когда разрешается писать несколько операторов в строке, переносить текст с одной строки на другую почти в любом месте, Фортран предполагает строгую форму записи.

В одной строке записывается не более одного оператора. Формат строки тоже строго определен, а ее длина не может превышать 72 символов. Такие требования были установлены первым компилятором Фортрана и, несомненно, облегчали ему работу. Размер строки определялся использованием для ввода программы в компьютер стандартных 80-колонных перфокарт, на каждую из которых «набивалась» одна строка. Такие перфокарты (рис. 1.5) были изобретены еще в конце XIX века основателем IBM Германом Холлеритом и поэтому особо почитаются этой фирмой.

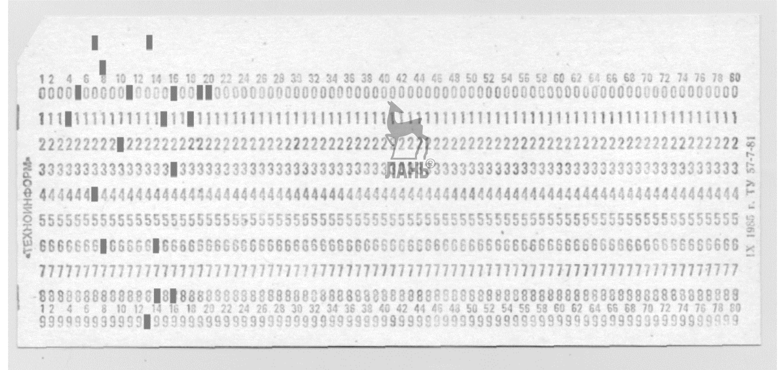


Рис. 1.5. 80-колонная перфокарта. Каждая колонка содержит пробивки кода одного символа. На карту нанесен текст оператора Фортрана:

```
10 DO 20 I=1, 100
```

Перенос текста программы на перфокарты обычно выполнялся не самим программистом, а оператором. Чтобы уменьшить число ошибок при перфорации программист записывал программу на специальных бланках (рис. 1.6). Каждая строка бланка, соответствующая одной перфокарте, разбита на несколько полей в соответствии с правилами записи программ на Фортране:

- Позиции с 1-й по 5-ю отводятся для размещения метки оператора. В роли меток используются целые числа без знака. Хорошим стилем считается размещать метки в программе по возрастанию, тогда их легче найти при чтении. Если позиция 1 содержит букву «С» (от «comment»), то строка является комментарием.

- Если в колонке 6 записан символ, отличный от пробела, то данная строка является продолжением предыдущей. Обычно в колонке продолжения ставят либо номер строки продолжения, либо какой-то выбранный символ, например «*».
- Оператор Фортрана записывается в позициях с 7 по 72. Он не обязан начинаться в седьмой колонке, но очень многие программисты долгое время игнорировали возможность использования отступов в фортрановских программах.
- Колонки 73–80 не обрабатываются компилятором Фортрана и могут использоваться, например, для нумерации перфокарт.

ЛПИ им. М. И. Калинина
Вычислительный центр
Программист Свердлов
Кафедра ТММ
Шифр М51
Дата 26.01.77
Лист 1 Листов 1
ЕС ЭВМ ФОРТРАН
Метка | Оператор
1 5 6 7 10 20 30 40 50 60 70 72 73 80

```
C     ПРИМЕР ПРОГРАММЫ НА ФОРТРАНЕ
      PROGRAM HELLO
      PRINT 10
   10 FORMAT(13HHELLO, WORLD!)
      END
```

Рис. 1.6. Программа «Hello, World!» на бланке для программирования на Фортране

Следуя давней традиции приводить для первого знакомства с языком программу, печатающую «Hello, World!», именно ее я подготовил для перфорации, записав на бланке. Чтобы вы лучше почувствовали дух самого старого языка программирования, использована древнейшая форма конструкции вывода: оператор `PRINT` в сочетании с оператором `FORMAT`. Две подряд идущие буквы «Н» в `FORMAT`е — не

опечатка. Просто строковая константа записана в так называемой холлеритовской форме с указателем длины. Начальная «H» — это первая буква в имени и фамилии Германа Холлерита (Herman Hollerith), а число 13 спереди, как вы догадываетесь, — количество символов в строке «Hello, World!». Я, признаться, до сих пор не понимаю, зачем потребовался такой странный способ представления строк.

Служебные слова, пробелы, описания

Двумя особенностями Фортрана, непривычными для современных программистов, являются отсутствие зарезервированных слов и незначащие пробелы. Например, служебное слово `IF` может использоваться в качестве имени переменной, а уж дело компилятора распознать, в каком смысле оно употреблено. Пробелы вообще игнорируются компилятором, за исключением пробелов внутри строковых констант. Например, запись оператора цикла:

```
DO 50 I = 1, 10
```

означающего десятикратное повторение следующих за ним операторов (вплоть до оператора, помеченного меткой `50`) при значениях переменной `I` от 1 до 10, для компилятора совершенно идентична записи вообще без пробелов:

```
DO50I=1,10
```

Описания переменных в программе на Фортране необязательны. Если нет явного описания, то тип переменной определяется по первой букве имени. Если это `I, J, K, L, M` или `N`, то переменная относится к целому типу (`INTEGER`), иначе — к вещественному (`REAL`). Очень коварное, как мы сейчас понимаем, правило.

А теперь снова рассмотрим оператор

```
DO 50 I = 1. 10
```

Вы видите разницу с предыдущими вариантами? На месте запятой теперь точка. Думаете, это приведет к ошибке при компиляции? Ничего подобного! В силу того, что пробелы незначащие, а описания необязательны, такая запись будет воспринята компилятором как (правильный!) оператор присваивания вещественной (в соответствии с первой буквой) переменной `DO50I` вещественного же значения `1.10` (она целая и десять сотых). Рассказывают, что именно из-за такой не обнаруженной вовремя ошибки в управляющей программе, написан-

ной на Фортране, завершилась неудачей первая попытка запуска американского космического аппарата к Венере.

Структура программы и распределение памяти

Программа на Фортране состоит из отдельных программных единиц — главной программы (`PROGRAM`), подпрограмм (`SUBROUTINE`, аналог процедур в Паскале), подпрограмм-функций (`FUNCTION`, аналог функций в Паскале и Си) и подпрограмм данных (`BLOCK DATA`).

Программные единицы могут транслироваться отдельно. Речь идет о *независимой компиляции*, когда при трансляции одной программной единицы компилятору недоступны сведения о других, а значит, компилятор не контролирует правильность вызова подпрограмм, не проверяет соответствие количества и типа фактических и формальных параметров.

Возможность разделения программы на независимые части оказалась очень ценной чертой Фортрана. Во-первых, это сделало язык пригодным для создания больших и очень больших программ: возможность разбиения программы на отдельно транслируемые модули-подпрограммы важна для организации разделения труда. Во-вторых, было разработано огромное количество подпрограмм для решения разнообразных, в первую очередь математических, задач. Такие тщательно тестированные и оптимизированные подпрограммы собраны в обширные библиотеки, обилие которых во многом и определило долголетие Фортрана. Лишь в конце 1970-х годов появились языки (Ада, Модула-2), в которых модульный механизм был принципиально совершенней фортрановского.

Параметры подпрограмм передаются в Фортране только по ссылке. Этот механизм подобен параметрам-переменным в Паскале или передаче адреса при вызове функции в Си. Изменение в подпрограмме формального параметра влечет и изменение фактического. Если фактическим параметром является константа или выражение, то их значение помещается в отдельную ячейку, ссылка на которую (ее адрес в памяти) передается в подпрограмму.

В Фортране используется *статическое* распределение памяти. Это означает, что каждой переменной или параметру подпрограммы назначается при компиляции одна ячейка памяти, и в течение всего времени выполнения значение хранится в единственном экземпляре в этой ячейке. Никаких действий по распределению памяти в ходе выполнения программы при этом не требуется. Такой механизм очень

эффективен и прост, но делает невозможным использование рекурсии.

Типы и структуры данных

В Фортране предусмотрены целые (INTEGER, разной разрядности), вещественные одиночной (REAL) и двойной (DOUBLE PRECISION) точности, комплексный (!) (COMPLEX) и логический (LOGICAL) типы. Как видно, — полный набор для решения вычислительных задач и полное отсутствие средств представления символьной информации.

Но на Фортране, как известно, программировались и компиляторы. Например, оптимизирующий компилятор Фортрана для компьютеров IBM/360, IBM/370, ЕС ЭВМ написан на Фортране. А компилятор — это программа, обрабатывающая текст. Для ввода текста в Фортране IV была предусмотрена возможность, считывая символы, помещать их в переменные целого или вещественного типа. Для этого требовалось указать специальный формат ввода.

Что касается структур данных, то предусмотрены только массивы (в Фортране IV размерностью до семи). Массивы, объявленные в подпрограмме, могут иметь переменный размер.

Управление последовательностью действий

Основными средствами организации управления в программе на Фортране являются операторы IF (примитивный арифметический и чуть более удобный логический), GOTO и циклы DO (аналог **for** в Паскале и Си). А поскольку вложенности операторов практически нет, составной оператор отсутствует, нет слова **else**, циклов с пред- и постусловием, то главным средством оказывается оператор перехода GOTO. Доля GOTO в программах на Фортране очень велика. Недовольство фортрановским стилем программирования с обилием операторов GOTO, провоцирующих хаотическую организацию управления, послужило одной из причин появления структурного программирования.

Ввод-вывод

В отличие от появившегося чуть позже Алгола-60 и некоторых современных языков, в которых средства ввода-вывода не являются частью языка, и, бывает, даже не специфицируются вместе с языком, в Фортран они были встроены с самого начала. Это сделало язык весьма практичным. Предусмотрены операторы ввода и вывода для

различных устройств, средства управления накопителями на магнитной ленте и т. п. Довольно странно с современных позиций выглядит задание формата вводимых и выводимых данных. Для этого предусмотрен специальный (невыполняемый!) оператор FORMAT, отголоски которого можно наблюдать в Си в виде первого аргумента printf.

Пример программы на Фортране

В качестве примера приведу подпрограмму, выполняющую сортировку по невозрастанию методом простых вставок массива вещественных чисел (листинг 1.1).

Я постарался написать ее в хорошем стиле. Использовал отступы, выделив начало внутреннего цикла с предусловием (два IF) и его конец (GOTO 10). Метки расставлены по возрастанию, в этом случае их проще находить при чтении большой программы.

Листинг 1.1. Пример программы на Фортране

```
C СОРТИРОВКА ПРОСТЫМИ ВСТАВКАМИ
      SUBROUTINE INSSORT(A, N)
      DIMENSION A(N)
      DO 40 I = 2, N
        X = A(I)
        J = I - 1
   10   IF(J) 30, 30, 20
   20   IF(X.GE.A(J)) GOTO 30
          A(J+1) = A(J)
          J = J - 1
        GOTO 10
   30   A(J+1) = X
   40 CONTINUE
      RETURN
      END
```

Небольшие пояснения к тексту. Параметрами подпрограммы являются массив A и его размер N. Тип величин A и N определен буквой: A — вещественный, N — целого типа. Оператор DIMENSION (неисполняемый) позволяет указать размер массива. Если массив и переменная, задающая его размер, — формальные параметры, то можно объявить массив переменного размера. Это очень ценное свойство, позволяющее создавать универсальные подпрограммы, не привязанные к размеру обрабатываемых массивов. Циклы DO мы уже обсуждали. В этом примере цикл заканчивается оператором 40 CONTINUE — пу-

стым оператором, который употребляют, во-первых, для большей ясности текста, а во-вторых, в связи с тем, что не всякий оператор может заканчивать цикл (таких особых случаев в Фортране было очень много, это в какой-то степени дух языка).

Для организации внутреннего цикла я вначале хотел записать

```
IF ( J .LE. 0 .OR. X .GE. A(J) ) GOTO 30.
```

Эта запись означает «если `J` меньше или равно (`.LE.`) 0 или `X` больше или равно (`.GE.`) `A(J)`, перейти на оператор с меткой `30`». Однако вспомнил, что короткая схема вычисления логических выражений не гарантируется, а значит, при `J=0` может проверяться вторая часть условия, и произойдет обращение к `A(0)`, которого нет, поскольку индексы в Фортране начинаются с 1. Впрочем, ничего страшного и в этом случае скорее всего не произойдет, просто будет обращение к ячейке памяти, предшествующей `A(1)`, но вне зависимости от ее содержимого условие все равно будет истинным. Тем не менее, я решил действовать строго и употребил два `IF`, получив, к тому же, возможность продемонстрировать и логический, и арифметический. Арифметический условный оператор `IF (J) 30, 30, 20` выполняет переход на метку `30`, если выражение в скобках меньше нуля, на метку `30`, если выражение равно нулю, или на `20`, если арифметическое выражение в скобках больше нуля.

Фортран 77

В семидесятые годы недостатки Фортрана стали очевидными. Уже появились более совершенные во многих отношениях языки, уже распространились и стали общепризнанными идеи структурного программирования. Не мог оставаться неизменным и Фортран. Был принят новый стандарт, получивший название Фортран 77. Фортран 77 расширяет предыдущий стандарт, сохраняя возможность транслировать старые программы компиляторами Фортрана 77. Дополнений довольно много. Назову, на мой взгляд, основные.

- Текстовый тип `CHARACTER` — символьные строки фиксированной длины.
- Структурные операторы `IF-THEN, ELSE IF, ELSE, END IF`.
- Ввод-вывод в свободном формате.
- Работа с файлами прямого доступа. Дополнительные средства для работы с файлами.

- Устранение ряда особых случаев и устаревших ограничений.
- Символические константы.
- Нулевые и отрицательные индексы массивов.

Фортран 77 не имел такой популярности, как предыдущие версии. К моменту появления компиляторов Фортрана 77 уже широко применялись Паскаль, Си, Бейсик, во многом более привлекательные для программистов. В СССР компиляторы Фортрана 77 использовались в 80-е годы в основном на мини-ЭВМ серии СМ.

Фортран 90 и Фортран 95

Работа над языком продолжалась. Третьим стандартом ANSI на язык Фортран стал Фортран 90. Внесенные в язык расширения весьма существенны. Вот основные:

- Свободная форма записи программы.
- Модернизированные управляющие операторы, в первую очередь цикл `DO`. Добавлен оператор `CASE`.
- Задание точности вычислений.
- Обработка массивов целиком. Например, можно записать `A=B*SIN(A)`, где `A` и `B` — массивы.
- Динамическое распределение памяти, указатели.
- Рекурсия.
- Типы данных, определяемые пользователем.
- Модули.

Фортран 90 является расширением Фортрана 77, и программа, написанная в соответствии со стандартом ANSI на Фортране 77, должна компилироваться транслятором Фортрана 90. Вместе с тем в стандарте Фортрана 90 некоторые элементы, сохранившиеся от первых версий языка, названы устаревшими. Это означает, что в следующих версиях они могут быть удалены. В число таких устаревших элементов попали, например, упоминавшиеся холлеритовские строковые константы, арифметический `IF`, возможность завершения цикла `DO` оператором, отличным от `CONTINUE`.

Фортран 95 имеет статус стандарта Международной Организации по Стандартизации (International Organization for Standardization, ISO). Добавления по сравнению с Фортраном 90 не слишком радикальны. Можно назвать новый оператор `FORALL`, введенный как альтернатива оператору `DO`. Интересны «чистые» функции (pure functions) — функ-

ции без побочного эффекта[3] и элементные функции — скалярные чистые функции скалярных аргументов. Очевидно, гарантии отсутствия побочных эффектов улучшают возможности оптимизации, что для Фортрана традиционно важно.

Наконец, впервые за сорок (!) лет из Фортрана удалены некоторые конструкции, объявленные устаревшими в предыдущей версии. В их числе холлеритовские строки с указателем длины. Пополнился список устаревших средств. Там по-прежнему арифметический IF, некоторые древние формы переходов и кое-что еще.

Фортран жив

Как видите, Фортран не забыт, появляются его обновленные стандарты. И хотя язык не входит в число остромодных, он сохраняет свое значение в сфере научно-технических расчетов. Такие компании как IBM, Microsoft и Intel считают своим долгом иметь в арсенале компиляторы Фортрана. Фортран находится в центре исследований и разработок систем для параллельных вычислений. Для суперкомпьютеров компиляторы Фортрана разрабатываются в числе первых.

Алгол-60

В 1958 году американская Ассоциация по вычислительной технике (Association for Computing Machinery, ACM) и европейское Общество по прикладной математике и механике (GAMM) создали совместный комитет по разработке стандарта международного алгебраического языка. Первоначально этот язык называли IAL (International Algebraic Language), затем он получил название Алгол (Algol, ALGOrithmic Language — алгоритмический язык). В состав комитета входил и Джон Бэкус — «отец» Фортрана. На заседании комитета, состоявшемся в мае 1958 года в Швейцарском федеральном техническом университете (Eidgenössische Technische Hochschule, ETH) в Цюрихе, был принят первый вариант такого языка. Позднее эту, в общем-то, предварительную версию стали называть Алгол-58.

В течение некоторого времени новый язык обсуждался и осмысливался. В частности, Бэкус пришел к выводу о необходимости формального описания Алгола. Основываясь на идеях американского лингвиста Ноама Хомского (Noam Chomsky), опубликовавшего в те

[3] Побочный эффект — изменение функцией значений параметров или глобальных переменных — затрудняет анализ и оптимизацию программы.

годы свои пионерские работы по теории языков, Бэкус предложил для описания Алгола формальную нотацию, получившую название БНФ — Бэкуса нормальная форма.

В январе 1960 года комитет по Алголу собрался в Париже в расширенном составе для доработки языка. Среди новых участников был датский астроном Петер Наур, предложивший переработанный вариант Алгола, синтаксис которого был описан им с помощью БНФ. С тех пор существует другая расшифровка аббревиатуры БНФ — Бэкуса — Наура форма. Принятый на конференции язык и называется Алгол-60.

В 1962 году спецификации Алгола были уточнены и опубликованы в так называемом «Пересмотренном сообщении». В «Модифицированном сообщении» в 1976 году сделан еще ряд поправок.

В Западной Европе и в СССР появление Алгола-60 было встречено с энтузиазмом, в то время как в Соединенных Штатах язык натолкнулся на серьезное противодействие, поскольку был конкурентом Фортрана, за которым стояли коммерческие интересы могущественной IBM.

Довольно быстро появились трансляторы Алгола. В числе первых были трансляторы, разработанные в СССР.

Уже весной 1962 года в космическом КБ С. П. Королева был запущен в эксплуатацию транслятор ТА-1, разработанный группой В. А. Степанова под руководством С. С. Лаврова для машины М-20. Это был первый отечественный транслятор с языка высокого уровня. Правда, входной язык ТА-1 был подмножеством полного Алгола и не включал самых трудных для реализации конструкций. Однако исключенными из языка оказались в основном те возможности, которые и в дальнейшем не нашли поддержки в языках программирования.

Годом позже, весной 1963 года заработал транслятор ТА-2 для той же машины М-20, созданный в Институте прикладной математики АН СССР (ИПМ) под руководством М. Р. Шура-Бура. В ТА-2 был реализован практически полный Алгол-60, включая рекурсию.

Третьим был разработанный под руководством А. П. Ершова в Новосибирске Альфа-транслятор. Работа над Альфа-системой автоматизации программирования началась уже с появлением Алгола-58. Входной язык Альфа-транслятора представлял собой расширенное подмножество Алгола. Особенностью Альфа-транслятора была глубокая оптимизация, позволившая формировать машинный код, лишь

на 30% по размеру и на 20% по скорости уступавший коду, полученному при программировании вручную. Альфа-транслятор обладал и мощными средствами отладки. Работа по Альфа-системе была завершена в 1964 году. Многие решения, примененные в этой системе, продолжают использоваться в современных оптимизирующих компиляторах.

Основные черты Алгола-60

Алгол, как и Фортран, ориентирован на численные расчеты. Это видно уже из его первоначального названия. Несомненно и то, что опыт использования Фортрана был принят во внимание при создании Алгола. Одну из ведущих ролей при разработке Алгола играл автор Фортрана Дж. Бэкус. Но Алгол не похож на Фортран.

В целом Алгол выглядит существенно совершенней, логичней и естественней Фортрана. В языке впервые появилось много новых для того времени, но совершенно естественных для современных языков средств.

Формальное описание

С самого начала при разработке Алгола была поставлена задача точной спецификации языка. При этом не имелся в виду никакой конкретный компьютер, на котором язык будет реализован. Более того, язык рассматривался не только как средство программирования вычислительных машин, но как формальная нотация для записи и публикации алгоритмов. Логичным следствием такого подхода явилось использование предложенной Бэкусом формальной нотации — БНФ, с помощью которой был определен синтаксис Алгола.

Вот как выглядит в виде формулы на БНФ определение идентификатора в Алголе, то есть последовательности букв и цифр, начинающейся с буквы:

```
<идентификатор> ::= <буква> | <идентификатор><буква> |
                    <идентификатор><цифра>
```

Это надо понимать так: идентификатор — это или буква, или результат приписывания буквы справа к тому, что уже является идентификатором, или идентификатор, к которому справа приписана цифра. Как видно, БНФ существенно рекурсивна, понятия в ней определяются через самих себя. Чтобы приведенная формула была

окончательно и однозначно понятна, необходимо определить, что такое `<буква>` и что такое `<цифра>`. С цифрой все просто:

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

А определение понятия `<буква>` вы теперь легко можете записать на БНФ сами, учитывая, что в Алголе разрешены большие и маленькие буквы латинского алфавита.

Формальное описание синтаксиса Алгола имело большое значение для всего последующего развития. Создатели большинства появившихся позднее языков предусматривали их формальное описание. Таким образом, появился аппарат, позволяющий однозначно и быстро отвечать на вопросы по правилам записи конструкций языка.

Очень скоро выяснилось, что формальная БНФ-спецификация — это не только свод правил. Она может служить непосредственным источником при разработке транслятора, структура которого однозначно следует из формального описания языка. Именно такой подход и будет использоваться в этой книге. Программу-транслятор мы будем писать прямо по описанию языка с помощью несколько модернизированной БНФ.

Формальное описание синтаксиса может быть исследовано формальными же методами. Совсем нетрудно себе представить, как совокупность формул БНФ поступает в качестве исходных данных на вход некоей программы. Эта программа, в зависимости от поставленной задачи, либо дает ответы на те или иные вопросы относительно этих формул, либо тем или иным образом преобразует эти формулы, либо, например, создает другую полезную программу — транслятор или хотя бы его часть.

Внешняя форма программы

Формат записи программы на Алголе свободный, то есть разделение на строки не играет существенной роли. Используемые в Алголе служебные слова в печатном тексте обычно воспроизводятся жирным шрифтом, а в рукописном — подчеркиваются. Способ, каким они записываются при вводе программы, должен был выбрать разработчик транслятора, обеспечив однозначное соответствие символов *языка конкретного представления* и *эталонного языка*. Чаще всего такие слова записывали в апострофах. В Алголе нет зарезервированных слов. Да и самого термина «служебное слово» нет. Запись **`begin`** или `'BEGIN'` — это просто разделитель и не имеет никакого отношения

к идентификаторам. Никто не запрещает назвать, например, переменную словом `begin`.

```
                        ×  ×
                                   TP НСЛЯТОР МЭИ-3
'BEGIN'
  'COMMENT'DVIGATEL.,[1-11]
  'REAL'MK,SK,S,NO.,[2-12]
  'ARRAY'M,N(/1:20/).,[3-11]
  'INTEGER'SPE,GR,NOM,I.,[4-14]
  POVTOR:INPUT(1,SPE,GR,NOM).,[5-27]
  OUTPUT(3,'(I'),SPE,GR,NOM).,[6-25]
  INPUT(1,MK,SK,NO).,[7-18]
  OUTPUT(3,'(E'),MK,SK,NO).,[8-23]
  OUTPUT(3,'(T'),'(:/')).,[9-19]
  S:=0.,[10-4]
  'FOR'I:=1'STEP'1'UNTIL'20'DO'
  'BEGIN'
    S:=S+.05.,[11-19]
    M(/I/):=2×MK/(SK/S+S/SK).,[12-22]
    N(/I/):=NO×(1-S)
  'END'.,[13-15]
  OUTPUT(3,'(E'),M,N).,[14-18]
  'FOR'I:=1,2,3'DO'OUTPUT(3,'(T'),'(:/')).,[15-28]
  'IF'KEY(2)'THEN''GOTO'POVTOR
'END'DVIGATEL
```

Рис. 1.7. Распечатка Алгол-программы

Ввод-вывод

Процедуры ввода-вывода в Алголе не были определены. Разработчики опасались, что не удастся избежать ориентации на возможности конкретных машин. Все решения относительно ввода-вывода возлагались на создателей трансляторов. Много лет спустя, уже в 1976 году в «Модифицированном сообщении» процедуры ввода-вывода включили в спецификацию языка, но было поздно. К тому времени разработка трансляторов Алгола уже фактически прекратилась.

Считается, что неопределенность с вводом-выводом нанесла ущерб распространению Алгола, поскольку делала несовместимыми различные системы. Затруднялась подготовка учебных пособий. Последнее осложняет и мою задачу. Я не могу в качестве примера привести программу «Hello, World!». На входном языке разных трансляторов она будет выглядеть по-разному. Вместо этого на рисунке 1.7 вы можете видеть реальный листинг небольшой учебной Алгол-

программы, распечатанный транслятором МЭИ-3 на машине «Минск-22». Можно разглядеть процедуры ввода-вывода (INPUT, OUTPUT), реализованные в этом трансляторе. Транслятор сам размещал операторы в распечатке, нумеруя их (в квадратных скобках справа). Обратите внимание на запись служебных слов в апострофах, замену индексных скобок «[» и «]» на «(/» и «/)», точки с запятой «;» на «.,». Все это особенности конкретного представления, допустимые описанием языка.

Структура программы и распределение памяти

Принятые в Алголе решения относительно структуры программы были новы и оказали значительное влияние на последующие языки. Основным понятием является *блок* — конструкция вида

```
begin <описания> ; <операторы> end.
```

Если описания отсутствуют, блок превращается в составной оператор. Если описания есть, они действительны в пределах данного блока. Вся программа также представляет собой блок. Блок является оператором. Тело процедуры представляет собой оператор. Блоки могут быть вложены произвольным образом. Собственно, последний тезис и так следует из предыдущих.

Если не иметь в виду модули, то по сути речь идет о блочной структуре общего вида. Такая общность и гибкость блочной структуры, с одной стороны, является мощным средством, а с другой — создает трудности при реализации, приводит к необходимости осуществлять дополнительные действия при выполнении программы, снижая тем самым ее эффективность.

В Алголе нет раздельной компиляции. Программа транслируется целиком. Это является серьезным препятствием для использования языка в больших проектах. Правда, некоторые трансляторы Алгола предусматривали независимую трансляцию процедур и их последующее использование как «внешних».

Описания переменных и массивов в Алголе обязательны. Память под переменные и массивы, описанные в начале каждого блока, динамически выделяется в момент начала выполнения этого блока. Объем памяти заранее (при трансляции) неизвестен, поскольку размер массивов, описанных в блоке, может задаваться с помощью переменных (описанных в одном из охватывающих блоков). Но поскольку память, выделенная последнему по ходу выполнения блоку, освобождается первой, речь идет хотя и о динамическом, но подчи-

ненном определенной (стековой) дисциплине, порядке распределения памяти.

Процедуры

Процедуры и процедуры-функции Алгола достаточно похожи на процедуры и функции в современных языках. Но в Алголе есть механизм, аналога которому вы не найдете, — передача параметров по наименованию.

Такая передача параметров означает текстуальную подстановку выражения фактического параметра вместо формального во все операторы тела процедуры с последующим выполнением тела процедуры так, как если бы оно находилось в месте вызова процедуры. Это правило позволяет записать на Алголе удивительные с нынешней точки зрения вещи.

Приведу пример. Ниже записана процедура-функция, вычисляющая сумму.

```
real procedure Sum(i, n, a);
value n;
real a; integer i, n;
begin real S;
   S := 0;
   for i := 1 step 1 until n do
      S := S + a;
   Sum := S;
end
```

Может показаться, что здесь что-то не так, что сумма всегда будет равна a×n, и суммировать бессмысленно. Но дело в том, что параметры передаются процедуре по наименованию (передача по значению указывается явно с помощью списка значений со словом **value**), поэтому при вызове `Sum(k, m, 1/k)` будет вычислена сумма первых m членов гармонического ряда, то есть:

$$\sum_{k=1}^{m}\frac{1}{k}=1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{m}.$$

Если же вычислить `Sum(i, 100, a[i])`, то получим сумму ста элементов массива a, поскольку вместо формального параметра a в текст оператора `S := S + a` будет подставлено `a[i]` и получится и будет выполняться `S := S + a[i]`.

Этот трюк известен как «прием Йенсена». В более поздних языках от передачи параметров по имени отказались как от сложного в реа-

лизации и неэффективного механизма, не дававшего особых преимуществ по сравнению с передачей по ссылке. Ну а сужение возможностей для трюкачества — скорее достоинство, а не изъян языка.

Передача параметра по наименованию реализуется в компиляторе с помошью *санков* (thunk — термин, введенный П. Ингерманом) — подпрограмм без параметров, генерируемых компилятором и вычисляющих значение параметра при каждом обращении к нему. Указатель на санк передается в процедуру.

Процедуры в Алголе могут вызываться рекурсивно, хотя не во всех трансляторах с Алгола рекурсия поддерживалась.

Типы и структуры данных

Набор типов в Алголе небогат. Это **`real`**, **`integer`** и **`Boolean`**. Имеются строки, но их употребление крайне ограничено, присутствуют, по сути, лишь строки-константы. Обеспечивается контроль соответствия типов. Тип выражения определяется статически, то есть на стадии компиляции. Разрешено присваивание целой переменной вещественного значения, которое в этом случае округляется. А вот смешивание логических и арифметических значений недопустимо.

Единственная предусмотренная в Алголе структура данных — это массив. Как уже говорилось, массивы могут быть переменного размера.

Управление последовательностью действий

Алгол — это первый язык, в котором появился достаточный набор управляющих операторов, позволяющий писать программу, обходясь без **`go to`**. За десять лет до распространения идей структурного программирования Алгол уже был «структурным» языком. Собственно говоря, само структурное программирование, которое мы обсудим далее, и дает ответ на вопрос, можно ли и следует ли писать программы на языках подобных Алголу без **`go to`**.

Тем не менее, кроме конструкции **`if`** – **`then`** – **`else`**, перешедшей в Паскаль и Си практически в неизменном виде, и богатого возможностями цикла **`for`**, в Алголе имеются и весьма изощренные разновидности переходов и, так называемые именующие выражения, значением которых является метка, также используемая для перехода. До наших дней эти конструкции переходов не дошли.

Пример программы на Алголе-60

В качестве еще одного примера приведу, как и для Фортрана, программу сортировки вещественного массива (листинг 1.2). Чтобы увидеть особенности блочной структуры, рассмотрим программу целиком. Для ввода и вывода я использовал процедуры, рекомендованные Модифицированным сообщением.

Листинг 1.2. Программа на Алголе-60

```
begin
   comment сортировка вставками;
   procedure InsSort(a, n);
   value n;
   array a; integer n;
   begin integer i, j; real x;
      for i := 2 step 1 until n do begin
         x := a[i]; a[0] := x;
         j := i;
         for j := j-1 while x<a[j] do
            a[j+1] := a[j];
         a[j+1] := x
      end
   end InsSort;

   integer n;
   ininteger(1, n);
   begin comment новый блок, чтобы определить массив;
      array x[0:n]; integer i;
      for i := 1 step 1 until n do
         inreal(1, x[i]);
      InsSort(x, n);
      for i := 1 step 1 until n do
         outreal(2, x[i]);
   end конец внутреннего блока;
end
```

Хотя в целом, надеюсь, текст понятен, отмечу некоторые моменты. Имеются два вложенных блока. Второй потребовался только для того, чтобы дать описание массива переменного размера. Величина `n`, задающая число сортируемых элементов, описана и введена во внешнем блоке.

Нижняя граница индексов массива взята равной 0, чтобы, используя в процедуре «барьер» (`a[0]:= x`), избавиться от составного условия во внутреннем цикле, надеясь на некоторое ускорение работы

программы и заодно обойти вопрос о способе вычисления составного условия.

В спецификации формальных параметров **array** a означает **real array** a. Размер и размерность, как видите, в такой спецификации не указываются. Массив x также считается вещественным. Параметр n передается в процедуру по значению, поскольку упомянут в списке значений (**value** n).

В процедуре сортировки использованы две разновидности циклов, точнее, цикл с двумя разновидностями элементов. Внешний цикл — с элементом списка цикла типа арифметической прогрессии; внутренний — с элементом типа пересчета. Все циклы в Алголе были обязаны иметь параметр (управляющую переменную) и, соответственно, начинаться с **for**. Привычный нам цикл с предусловием (**while**) еще не родился.

Цифры 1 и 2 в обращениях к процедурам ввода-вывода означают номера так называемых каналов ввода-вывода. В примере они условны.

Повторение названия процедуры после завершающего **end** не обязательно. Просто в этом месте в Алголе разрешен комментарий, и этим часто пользуются.

Судьба Алгола

Как уже говорилось, Алгол-60 оказал очень большое влияние на развитие языков программирования высокого уровня. К числу алголоподобных языков относятся Паскаль, Модула-2, Ада, Оберон. Черты Алгола можно увидеть в Си и в Яве.

Основные достижения Алгола: строгое определение, блочная структура, обязательность описаний, развитые средства управления последовательностью действий, рекурсия.

Алгол долгое время использовался как объект и инструмент в исследованиях по программированию и методам трансляции, служил основным языком публикации алгоритмов.

В качестве языка программирования он имел значительное распространение в Европе. В СССР был в течение 60-х и первой половины 70-х годов основным языком программирования, значительно опережая Фортран. В США Алгол не вышел за пределы университетов.

С распространением в СССР ЕС и СМ ЭВМ, являвшихся клонами американских IBM/360/370 и PDP-11, Алгол был вытеснен Фортра-

ном, что в известной мере стало шагом назад. В дальнейшем на смену пришли другие языки. Широкое распространение уже в новые времена Паскаля в нашей стране, вероятно, можно объяснить существованием и передачей от поколения к поколению алгольной традиции.

Примерно с начала 1980-х годов использование Алгола-60 прекратилось. Кроме прочего, это было обусловлено и его недостатками: отсутствием средств обработки нечисловой информации, нестандартизованным вводом-выводом, отсутствием модульности.

Кобол

В те же годы, когда появились Фортран и Алгол, — языки для программирования научно-технических задач, была осознана и потребность в средствах разработки учетно-бухгалтерских систем. Компьютеры пришли в бизнес. Был создан язык Кобол (COBOL, COmmon Business Oriented Language — универсальный язык, ориентированный на бизнес). Произошло это в США, да и где же еще могло произойти.

В нашей стране потребность в подобных языках в те годы еще не сформировалась. Да и бизнеса (как и еще некоторых важных вещей) в СССР не существовало. Вычислительные машины довольно долго не использовались для повседневных нужд предприятий. Хотя отечественные ЭВМ появились не намного позже американских и европейских, выпускались они в небольших количествах (к примеру, машина М-20, для которой создали несколько трансляторов Алгола, была изготовлена в количестве всего 20 экземпляров) и применялись для выполнения расчетов, связанных с техническими разработками и научными исследованиями в основном оборонной направленности.

Кобол разработан в 1960 году комитетом «КОДАСИЛ» (CODASYL, Conference on Data Systems Languages — конференция по языкам систем обработки данных), организованным министерством обороны США. В состав комитета вошли представители ведущих американских компьютерных фирм. Инициировала создание этого комитета Грейс Хоппер.

Класс задач, на которые ориентирован Кобол (экономические задачи, как у нас было принято говорить), не предполагает сложных вычислений. Тут не нужен синус. Зато необходимо обрабатывать большие объемы данных, формировать должным образом оформленные выходные документы.

Кобол содержит развитые средства структурирования, обработки, сортировки и редактирования данных большого объема, хранящихся в файлах. Имеются средства генерации отчетов.

Другая черта Кобола — использование формы записи, приближенной к естественному английскому языку. Считается, что это облегчает понимание текста программ на Коболе людям, не имеющим математической подготовки. Для того чтобы операторы языка больше походили на английские фразы, предусмотрено даже использование так называемых шумовых слов, которые никак не влияют на смысл программы и не нужны для разделения ее частей, а применяются исключительно для большей «естественности». Даже обычные арифметические операции обозначаются английскими словами. Однако есть мнение, что в действительности это не приносит пользы. Как замечает Дэвид Баррон, «понятными делает программы *структура*, а это как раз то, чего не хватает программам, написанным на Коболе» [Баррон, 1980].

Программы на Коболе громоздки и многословны (листинг 1.3). В этом отношении Кобол — истинно американский язык. Подмечена склонность американцев употреблять слова там, где другие предпочитают условные обозначения. Даже вместо дорожных знаков в Штатах иногда используют надписи. У нас будет повод вспомнить об этой американской черте и при разговоре о других языках.

Вот пример оператора, который выглядит как обычная фраза:

```
ADD GIN TO VERMOUTH GIVING MARTINI.
```

(Добавь джина к вермуту, чтобы получить мартини.) Вряд ли этот рецепт имеет отношение к той программе, в которую его поместили. Наверное, немало шутников острило таким образом.

Листинг 1.3. «HELLO, WORLD» на Коболе

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.     HELLOWORLD.
000300 DATE-WRITTEN.   02/05/96        21:04.
000400*      AUTHOR    BRIAN COLLINS
000500 ENVIRONMENT DIVISION.
000600 CONFIGURATION SECTION.
000700 SOURCE-COMPUTER. RM-COBOL.
000800 OBJECT-COMPUTER. RM-COBOL.
000900
001000 DATA DIVISION.
001100 FILE SECTION.
```

```
001200
100000 PROCEDURE DIVISION.
100100
100200 MAIN-LOGIC SECTION.
100300 BEGIN.
100400    DISPLAY " " LINE 1 POSITION 1 ERASE EOS.
100500    DISPLAY "HELLO, WORLD. " LINE 15 POSITION 10.
100600    STOP RUN.
100700 MAIN-LOGIC-EXIT.
100800    EXIT.
```

Кобол получил широкое распространение в Соединенных Штатах. Это было обусловлено не в последнюю очередь тем, что правительство США требовало наличия транслятора с Кобола на любом компьютере, приобретаемом за государственный счет.

В нашей стране Кобол не получил большого распространения. В момент его появления потребности в таком инструменте у нас, по сути, не было. Когда сформировалась потребность, появились другие средства, пригодные для решения экономических задач: универсальные языки, системы управления базами данных. Тем не менее, для ряда отечественных машин были разработаны трансляторы с русского варианта Кобола (с русскими словами взамен английских). А как же иначе? Ведь обилие английской лексики никак не могло способствовать лучшему пониманию программ людьми, говорящими по-русски, скорее наоборот. Интерес к Коболу проявили, например, в киевском Институте Кибернетики, которым руководил академик В. М. Глушков, всегда уделявший большое внимание вопросам использования вычислительных машин в планировании и управлении.

Две попытки объять необъятное

Появление к началу 1960-х годов большого числа специализированных языков и успешное распространение некоторых из них породили мечты о создании универсального языка, пригодного для решения самых разнообразных задач. Может быть, даже языка на все времена. Такие попытки и в самом деле были предприняты.

ПЛ/1

Programming Language number One, сокращенно PL/I, — язык программирования номер один — таково полное оригинальное название этого языка, немало говорящее о замыслах его создателей.

ПЛ/1 разработан и реализован в течение 1963–66 годов. Разработку языка вел комитет, созданный компанией IBM и ассоциацией пользователей компьютеров IBM. Кроме специалистов IBM в комитет вошли также представители ряда американских промышленных компаний. Работы по языку ПЛ/1 велись в рамках создания серии компьютеров IBM/360. Возглавлял разработку Джордж Рэдин (George Radin). Вот как он писал о поставленной задаче: «Комитет стремился к тому, чтобы язык мог удовлетворить программистов различных уклонов: программистов научных и коммерческих задач, программистов задач, решаемых в реальном масштабе времени, и программистов систем, а также к тому, чтобы как начинающие, так и опытные программисты могли использовать возможности языка в соответствии со своим уровнем» [Рэдин, 1967].

Поставленная задача была выполнена, хотя предварительно намеченные сроки окончания работы и были нарушены. Был создан действительно мощный и универсальный язык. Однако, как показал дальнейший ход событий, подход, примененный при создании ПЛ/1, оказался все же неудачным.

Главным недостатком этого языка считают чрезмерную сложность, точнее громоздкость, поскольку трудных для понимания концепций он, в общем-то, не содержит, но количество заложенных в язык разнообразных возможностей необычайно велико. Полная документация по ПЛ/1 для ЕС ЭВМ занимала свыше 2000 страниц.

Знакомясь с ПЛ/1, нетрудно обнаружить в нем черты Фортрана, Алгола-60 и Кобола. Неудивительно. Эти три языка в годы, когда создавался ПЛ/1, были самыми известными и популярными, и комитет, вырабатывавший спецификации ПЛ/1 и работавший, к тому же, в обстановке спешки, просто позаимствовал многое из этих языков. Из Фортрана — независимую компиляцию подпрограмм, способы передачи параметров, элементы форматного ввода-вывода. Из Алгола — блочную структуру и управляющие операторы. Из Кобола — ввод-вывод записей и шаблоны.

ПЛ/1 иногда называют языком-оболочкой, поскольку он представляет собой обширное хранилище готовых средств решения задач из различных предметных областей.

Примеры программ на ПЛ/1

Приведу два примера. «Hello World!» позволит увидеть законченную программу и простейшие средства вывода, а программа сортировки — организацию процедур, массивы и управляющие операторы.

```
HELLO: PROCEDURE OPTIONS(MAIN);
   PUT SKIP LIST('HELLO WORLD!');
END HELLO;
```

Как видите, главная (`MAIN`) процедура `HELLO`, с которой начинается выполнение программы, помечена меткой, служащей названием этой процедуры. Зачем требуется писать слово `OPTIONS`, я за несколько лет программирования на ПЛ/1 так и не узнал. Утешаюсь тем, что, как говорят, человека, знающего ПЛ/1 целиком, просто не существует. В примере использован вывод (`PUT`), управляемый списком (`LIST`), — самый простой из предусмотренных вариантов. Слово `SKIP` означает, что перед печатью произойдет переход на новую строку. Повторение названия процедуры после оператора `END` (`END` — оператор!) необязательно. А вот все три точки с запятой, имеющиеся в этом примере, — нужны. Точка с запятой не разделяет операторы, а завершает каждый из них (`PROCEDURE` — тоже оператор).

Процедуры в ПЛ/1 могут компилироваться независимо.

Теперь сортировка (листинг 1.4). На этот раз — пузырьковая.

Листинг 1.4. Процедура сортировки на языке ПЛ/1

```
/* ПУЗЫРЬКОВАЯ СОРТИРОВКА /
B_SORT: PROC (A, N);
   DECLARE A();
   DO I = 2 TO N;
      DO J = N TO I BY -1;
         IF A(J) < A(J-1) THEN DO;
            X = A(J-1); A(J-1) = A(J); A(J) = X;
         END;
      END;
   END;
END B_SORT;
```

Я предпочел бы назвать эту процедуру `BUBBLE_SORT`, но правилами ПЛ/1 запрещено использовать для процедур имена длиннее 7 символов — ограничение, связанное с особенностями операционной системы IBM/360. Локальные идентификаторы могут иметь длину до 31 символа.

Длинное слово PROCEDURE, как и некоторые другие слова, разрешается сокращать.

Вместо длины массива-параметра переменного размера записывается звездочка. Продолжает действовать плохое фортрановское правило определения типа по первой букве, если нет явной декларации. По этому правилу A и X имеют атрибуты DECIMAL FLOAT — десятичные с плавающей точкой, а I и J — BINARY FIXED — двоичные с фиксированной точкой. При желании можно задать точность представления, как тех, так и других, но если не указывать, то действуют атрибуты точности, принимаемые по умолчанию.

Для организации циклов используются операторы DO, а для группирования операторов — так называемые группы DO. И те, и другие употребляются в паре с END.

Основные черты ПЛ/1

- Многообразие встроенных типов данных: десятичные, двоичные, с фиксированной и плавающей точкой регулируемой разрядности, символьные и битовые строки постоянной и переменной длины, шаблоны и т. д.
- Допустимы почти любые преобразования типов, которые выполняются неявно. Правила преобразований очень запутаны, и запомнить их практически невозможно. При этом отсутствует контроль соответствия типов, что делает язык опасным. Попасть в ловушку очень легко. К примеру, при вычислении безобидного с виду выражения 2/3+25 возникает переполнение, а при такой записи 002/3+25 — не возникает!
- Ключевые слова не зарезервированы. Можно, к примеру, написать

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

- Предусмотрены встроенные средства ввода-вывода многочисленных разновидностей, включая ввод и вывод для файлов последовательного, прямого и индексно-последовательного доступа.
- Способы организации данных — массивы, в том числе переменного размера, подмассивы, структуры.
- Достаточные средства управления последовательностью действий.
- Разнообразные варианты распределения памяти: статическое, автоматическое, динамическое. Возможность наложения разнотипных

данных. Указатели, не привязанные к типу данных, на который они указывают.
- Обработка исключительных ситуаций (прерываний).
- Независимая трансляция процедур. Рекурсивные процедуры должны отмечаться явно.
- Средства параллельного выполнения процедур и операций ввода-вывода.
- Использование принципа умолчания. Авторы языка понимали, что он получается очень сложным, и постарались смягчить проблему. Принцип умолчания позволяет пользоваться языком, зная лишь какую-то его часть. Если программисту не известны все атрибуты описания данных, он может указывать лишь некоторые, а значения других будут приняты по умолчанию. Однако не всегда значения, предусмотренные по умолчанию, совпадают с теми, на которые рассчитывал программист.

Сложность языка ПЛ/1 создает трудности как при освоении, так и при реализации. Конструкция языка такова, что разработка компилятора для него становится трудной и трудоемкой задачей. Недаром вначале были выпущены компиляторы для подмножества языка. Вследствие трудности реализации и ориентированности на архитектуру операционной системы OS/360 язык ПЛ/1 был долгое время доступен лишь на компьютерах IBM и совместимых с ними.

Компиляторы ПЛ/1 работали медленно, создавая при этом медленные и объемные машинные программы, уступавшие в эффективности программам, полученным трансляцией с Фортрана.

Несмотря на свои недостатки, ПЛ/1 получил в 1970-е годы немалое распространение. В дальнейшем с появлением персональных компьютеров про него стали забывать. Вначале просто не хватало аппаратных ресурсов, чтобы реализовать такой громоздкий язык на персоналках, а затем на первый план вышли другие языки. Известны, однако, некоторые проекты, в которых ПЛ/1 использовался (в основном для переноса старых программ на новые платформы) в 1990-е годы. Компиляторы ПЛ/1 имеются на всех основных программно-аппаратных платформах.

В СССР ПЛ/1 был принят с немалым энтузиазмом. Воспитанные в алгольной традиции программисты, к тому же проникшиеся идеями структурного программирования, с удовольствием оставляли неуклюжий и неструктурный Фортран. Знанием ПЛ/1 можно было даже

щегольнуть. Понимание недостатков языка и ненадежности многих его конструкций пришло позже. А в конце 70-х и в 80-е годы XX века ПЛ/1 был одним из основных языков, применяемых на ЕС ЭВМ.

Алгол-68

Другой попыткой создания универсального языка в 1960-е годы стала разработка Алгола-68. После публикации *Пересмотренного сообщения об Алголе-60* в 1962 году его дальнейшая судьба была передана в руки Рабочей группы 2.1 ИФИП — Международной федерации по обработке информации (International Federation for Information Processing, IFIP). В состав группы входили как авторы Алгола-60, так и новые участники. Задачей группы было выработать подмножество Алгола-60, создать спецификации средств ввода-вывода и разработать новый язык, который должен был стать наследником Алгола-60.

От участников группы поступило много предложений по развитию языка. В мае 1965 года Никлаусу Вирту (Niklaus Wirth) было поручено обобщить эти предложения, объединив их в единый проект. Предварительный вариант Н. Вирта, как вспоминал в дальнейшем Тони Хоар[4], участник группы, произвел на него сильное впечатление. Это был язык, в котором были устранены недостатки Алгола-60 и добавлены некоторые новые возможности, удобные и безопасные, которые к тому же могли быть легко и эффективно реализованы. Хоар и ряд других участников подключились к окончательной доработке проекта. Ко времени очередного заседания группы в Сент-Пьер де Шартез в октябре 1965 года это был проект совершенного и практичного языка. (В дальнейшем этот язык был реализован на IBM/360 и получил в честь Н. Вирта название Algol W.)

[4] Чарльз Энтони Ричард Хоар (Charles Antony Richard Hoare), которого друзья называют Тони, — профессор Оксфордского университета и сотрудник Microsoft Research, авторитетнейший специалист в компьютерном деле, лауреат премии Тьюринга (1980), один из основоположников структурного и объектно-ориентированного программирования. В 1959 году, возможно во время стажировки в МГУ, изобрел алгоритм быстрой сортировки. В 60-е годы — сотрудник лондонской фирмы Elliott Bros. Ltd. За выдающиеся заслуги Королевой Великобритании Ч. Э. Р. Хоару присвоено рыцарское звание, теперь его титул Sir Tony Hoare. В литературе на русском языке с именами Ч. Э. Р. Хоара много путаницы. То он Т. Хоар, то К. Хоар или Ч. Хоар, а иногда вообще Хоор. Все это один и тот же человек, имейте в виду.

Однако на заседании, к которому был подготовлен проект Н. Вирта, комитет по Алголу в первую очередь рассмотрел другой проект, предложенный Аадом ван Вейнгаарденом (Aad van Wijngaarden), одним из авторов Алгола-60. Это был, как пишет Хоар, неполный и непонятный документ, определявший весьма амбициозный язык, не соответствовавший первоначальным предложениям. Несмотря на возражения Хоара, именно этот проект и был принят для дальнейшей работы.

Последовала серия доработок. С каждым разом проект становился сложнее, спецификации радикально менялись от одного варианта к другому. В проекте было большое количество ошибок, исправлявшихся в последний момент, сроки затягивались. Хоар с сожалением отмечает, что многие члены группы не только не стремились упростить и без того сложный язык, но, наоборот, настаивали на добавлении усложняющих его возможностей.

Наконец, 20 декабря 1968 года на заседании в Мюнхене язык был принят и получил название Алгол-68. Несколько членов группы направили ИФИП отчет с особым мнением, но отчет этот был отвергнут.

Алгол-68 — сложный язык. Но его сложность имеет несколько иную природу, нежели сложность ПЛ/1. Если ПЛ/1 просто объемен, то сложность Алгола-68 в том, что его механизмы носят предельно обобщенный характер. Авторы языка создали мощную, универсальную, но весьма изощренную конструкцию, для понимания которой требуется немалое напряжение. Даже для описания синтаксиса Алгола-68 формализма БНФ оказалось недостаточно. Алгол-68 определен с помощью грамматик А. ван Вейнгаардена, которые являются обобщением БНФ и, несомненно, не облегчают понимание языка.

Алгол-68 иногда характеризуют как язык-ядро. Он не содержит, в отличие от ПЛ/1, готовых рецептов решения разных задач, но обладает, как считается, свойством саморазвития. На основе имеющихся механизмов, используя, как любили говорить ценители Алгола-68, их ортогональную сочетаемость, можно создавать другие конструкции, как бы расширяя сам язык. В первую очередь, вероятно, имеются в виду возможности определять собственные типы данных и операции с ними.

Основные черты Алгола-68

- Блочная структура. Описания могут чередоваться с операторами. Инициализация при описании. Описания констант.
- Концепция видов (**mode**), которая вполне эквивалентна более привычной нам концепции типов. Каждый объект программы принадлежит некоторому виду. К определенному виду принадлежат, в том числе, процедуры и операции. Примитивные виды: **bool** (логический), **int** (целый), **real** (вещественный), **char** (символьный), а также **bits** (биты) и **bytes** (байты). Имеется вид **compl** (комплексный) с соответствующим набором операций. Предусмотрены длинные виды, например **long int** (целое удвоенной длины) и даже **long long int** (целое утроенной длины) и т. д., если позволяет реализация. Есть и короткие (**short**).
- Активно используемая концепция ссылок (**ref**). Привычное (единое) понятие переменной расщеплено на два: ссылка и значение. Это верно по сути, но усложняет восприятие языка. Может быть получена ссылка как на объект, созданный в куче, так и на локальный объект. Явное обозначение операции преобразования ссылки в значение (разыменование) отсутствует. Разыменование выполняется неявно в зависимости от контекста.
- Сборка мусора для освобождения блоков памяти в динамической области (куче), на которые больше нет ссылок из программы. Сборщик мусора практически избавляет программиста от необходимости заботиться об освобождении распределенной динамически памяти.
- Повышение роли операций. Многие библиотечные средства, которые мы привыкли получать в свое распоряжение как функции, в Алголе-68 являются одноместными операциями, которые изображаются идентификаторами. Например, запись `abs entier re z` означает абсолютную величину целой части вещественной компоненты *z* (предполагается, что *z* — комплексное). Впрочем, это же выражение можно записать и по-другому `abs(entier(re(z)))`, что, однако, не превращает операции в функции. Просто для указания порядка действий в выражении теперь расставлены (лишние) скобки. Язык предусматривает возможность определения программистом новых операций и переопределение (перегрузку) существующих одноместных и двуместных операций. Эта возможность,

нашедшая затем продолжение в языках Ада, Си++ и Си#, отсутствует, тем не менее, в языках Модула-2, Оберон и Ява.

- Развитые структуры данных. Массивы (мультизначения) могут быть переменного размера и с подвижными границами. Строки (**`string`**) — массивы из символов с подвижной верхней границей. Допустимы подмассивы (вырезки). Структурные виды (**`struct`**), аналогичные записям в других языках. Объединения (**`union`**) позволяют совмещать данные разных видов. Вид текущего значения объединения проверяется динамически. Есть возможность явной записи значений векторов и структур.
- Процедуры с параметрами и без параметров, возвращающие и не возвращающие значение. Параметры передаются по значению. Если параметр должен изменяться процедурой, его оформляют как ссылку. Параметр, не являющийся ссылкой, рассматривается в процедуре как константа.
- Достаточный набор управляющих конструкций. Условные: **`if`** – **`then`** – **`elif`** – **`else`** – **`fi`**. Выбирающие: **`case`** – **`of`** – **`else`** – **`esac`** (несмотря на похожие слова, отличается от аналогичной конструкции в Паскале). Циклы: **`for`** – **`from`** – **`by`** – **`to`** – **`while`** – **`do`** – **`od`**. Метки и переходы.
- Развитые средства ввода-вывода (обмена). Бесформатный, форматный и двоичный обмен. Файлы. Предусмотрены аварийные процедуры, которые вызываются в случае нештатных ситуаций при работе с файлами.
- Разбиение программы на единицы компиляции специально не предусмотрено. Но в сообщении о языке говорится о возможности в конкретной реализации независимой трансляции главной программы и процедур. Предполагается, что любой программе доступны средства, предоставляемые так называемым стандартным вступлением — стандартные виды, функции, операции. Предполагается также, что реализация дает возможность программисту создать библиотечное вступление, куда он может включить необходимые ему средства, которые будут доступны программе.
- Средства параллельного программирования с механизмом семафоров.
- Совмещение операторов и выражений. Любое предложение, вырабатывающее значение, в том числе и включающее операторы, может использоваться в выражении, тем самым создавая выражению

побочный эффект в виде изменения значений переменных в ходе вычисления выражения. И наоборот, выражения могут употребляться там, где разрешен оператор.

- Сокращения. Многие конструкции, определяемые «строгим» языком, могут быть сокращены.

  Например, предложение

```
begin ref real x = loc real;
   if y < 0 then x := -y else x := y fi;
…
end
```

  можно записать сокращенно:

```
(real x; ( y<0 | x := -y | x := y ); … )
```

  или еще короче:

```
(real x := ( y<0 | -y | y ); … )
```

  или вообще это:

```
(real x := abs y; … )
```

Примеры программ на Алголе-68

Сначала — законченная программа «Hello, World!», которая выглядит очень просто:

```
begin
   print(("Hello, World!", newline))
end
```

Заменив слова **begin** и **end** круглыми скобками, можно записать короче:

```
( print(("Hello, World!", newline)) )
```

При обращении к `print` двойные скобки в нашем случае обязательны, поскольку мы выводим два элемента: символьную строку и `newline` (это процедура). Внутренние скобки превращают фактический параметр `print` в запись вектора.

В качестве второго примера (листинг 1.5) рассмотрим процедуру сортировки вставками.

Листинг 1.5. Процедура сортировки на языке Алгол-68

```
# Сортировка вставками #
proc InsSort = (ref [] real a) void:
begin int k; real x;
```

```
    for i from lwb a + 1 to upb a do
       x := a[i];
       k := i-1;
       for j from k by -1 to lwb a while x < a[j] do
          a[j+1] := a[j];
          k := j-1
       od;
       a[k+1] := x
    od;
 end
```

Сортируется целиком массив вещественных чисел. Границы индексов фактического массива, используемого при обращении к этой процедуре, могут быть любыми. В описании параметра это указано пустыми скобками `[]`. В процедуре эти границы определяются с помощью операций **`lwb`** — нижняя граница и **`upb`** — верхняя граница. Обратите внимание, что нет отдельных описаний параметров циклов `i` и `j`. Упоминание после **`for`** описывает их неявно как целые. Область действия параметра `j` ограничена циклом, поэтому о его значении за пределами цикла говорить не приходится. Чтобы воспользоваться последним значением `j` пришлось использовать дополнительную переменную `k`.

Можно реализовать внутренний цикл по-другому. Вместо короткой схемы вычисления логических выражений используем условное выражение **`if`** `k>0` **`then`** `x<a[k]` **`else false fi`** (оно записано в сокращенной форме).

```
       while (k>0 | x<a[k] | false) do
          a[k+1] := a[k];
          k -:= 1
       od
```

Или еще один вариант. Используются метки (`while` и `exit`), переходы и сокращенная запись для **`if – then – fi`**.

```
       while:(k=0|exit);(x>=a[k]|exit);
          a[k+1] := a[k];
          k -:= 1;
       while;
       exit:
```

Конструкция `(k=0|exit)` — это сокращение **`if`** `k=0` **`then goto`** `exit` **`fi`**. Метки `while` и `exit` не имеют ничего общего с соответствующими служебными словами **`while`** и **`exit`**, которые должны

выделяться особо (подчеркиванием, жирностью, апострофами, точками). Обратите внимание на операцию, совмещенную с присваиванием в обоих вариантах. Запись `k -:= 1` (или `k` **`minusab`** `1`) означает `k := k-1`.

Уроки Алгола-68

Язык Алгол-68 без сомнения обладал выразительными средствами, достаточными для решения широкого круга задач. Конструкция языка выглядит стройно и непротиворечиво. Достигнутая «ортогональность» позволяет сочетать элементы языка самыми разнообразными способами.

Но ведь той же самой функциональности, как мы знаем, можно достичь куда более простыми средствами.

Создается впечатление, что в своем стремлении к обобщениям разработчики Алгола-68 упустили из виду основную цель, ради которой и создается язык. Цель эта — быть средством разработки программ, повседневным инструментом программиста. Инструментом полнофункциональным и универсальным, но в то же время возможно более простым. Язык должен способствовать формированию ясного стиля, не быть избыточным, а напротив, вводить разумные ограничения, способствующие повышению надежности и эффективности программ. Алгол-68 выглядит скорее как академическое упражнение, а не инструмент для повседневной работы программистов-практиков.

А ведь именно в те годы, когда создавался Алгол-68, были выдвинуты идеи структурного программирования. Стимулом к переосмыслению технологии послужило тогда стремление повысить надежность больших программных систем, уменьшить число ошибок, повысить производительность труда программистов. Принципы структурного программирования предполагают более жесткую дисциплину программирования, стандартизацию стиля, отказ от использования изощренных, трудных для понимания программистских приемов.

Идеология Алгола-68 во многом противоречит этим принципам. Язык стимулирует изощренный стиль, основанный на нетривиальном сочетании конструкций. Одним из источников появления неудовлетворительных по стилю программ является возможность совмещения выражений и операторов, что поощряет побочные эффекты, затрудняющие понимание программ. Не способствуют стандартизации стиля и альтернативные способы записи одних и тех же элементов и кон-

струкций. В языке содержатся возможности для разнообразных трюков.

Вспомним прием Йенсена. Хотя в Алголе-68 нет передачи параметров по имени, этот трюк может быть исполнен. Вот как выглядит процедура, которая способна суммировать почти все что угодно.

```
proc Summa = (ref int i, int n, proc real a) real:
begin real sum:=0;
   for k to n do
      i := k;
      sum +:= a
   od;
   sum
end;
```

Здесь слагаемое `a` передается как параметр процедурного вида. Ничего необычного в этом, в общем-то, нет, потому что процедурный тип имеется во многих языках. Но посмотрите, как эта процедура может быть вызвана:

`Summa(i, 100, real: a[i]);` или `Summa(k, m, real: 1/k);`

Формула, подставляемая в качестве третьего параметра, является процедурой вида **proc real**. Обратите внимание, что переменная `i` не может быть сделана параметром цикла, и пришлось выполнить дополнительное присваивание. Начальное значение `k` явно не задано, и по умолчанию оно равно 1. Запись `sum` перед **end** делает величину суммы значением замкнутого предложения (**begin** … **end**), это значение и возвращается функцией `Summa`.

Сложность Алгола-68, а также отсутствие поддержки со стороны ведущих компьютерных компаний привели к тому, что язык не получил широкого распространения. Несколько реализаций было осуществлено в Европе, сначала — подмножеств языка, а к середине 1970-х — и полного Алгола-68.

В СССР внимание на Алгол-68 обратили уже в момент его появления. Особый энтузиазм и настойчивость проявила группа из Ленинградского университета, возглавляемая Г. С. Цейтиным и А. Н. Тереховым. В 1969 году, еще до опубликования официального сообщения о языке, группа приступила к реализации Алгола-68. Лишь к 1976 году была завершена предварительная версия компилятора. ЕС ЭВМ, для которой предназначался компилятор, появилась только в 1974 году, поэтому анализирующая часть транслятора, напи-

санная на Алголе-60, работала на машине ODRA 1204, а синтезирующая, написанная на ассемблере, — на ЕС ЭВМ. Еще два года ушло на переписывание транслятора на Алгол-68, компиляцию новой версии с помощью старой, потом новой версии с помощью ее же самой и отладку на ЕС ЭВМ. В 1978 году компилятор был готов и передан для использования. Затем он был еще несколько модернизирован и с 1979 года использовался в Ленинградском университете для обучения студентов и в промышленных разработках. В 1988 году компилятор был перенесен на IBM PC-совместимые компьютеры.

Несмотря на то, что Алгол-68 не получил широкого распространения, он повлиял на ряд языков, появившихся позже. Черты Алгола-68 можно увидеть в языках Си и Си++. Создатель Си++ Бьерн Страуструп называл Алгол-68 одним из предшественников Си++. Даже русский алгоритмический язык из первого школьного учебника информатики, который многие считают похожим на Алгол-60 и Паскаль, в действительности содержит больше элементов именно из Алгола-68.

Интерактивное программирование для всех

Языки, которые мы уже обсудили, в момент своего появления были не очень легки для изучения. Специалисту, которому нужно было выполнить даже несложный расчет, если он сам собирался написать программу, приходилось преодолевать высокий «барьер вхождения», осваивая не слишком простые понятия языка программирования.

Вследствие сложной структуры языка, компиляция и последующая сборка программы занимала на компьютерах, ресурсы которых были скромны, немалое время. Ведь компилятор Фортрана или Алгола — это большая программа, которая, как правило, не помещалась целиком в память компьютера. Считывание с магнитной ленты или барабана частей компилятора, вступавших в работу последовательно, замедляло обработку, да и само число проходов компилятора (последовательных просмотров транслируемой программы от начала к концу) бывало больше двух десятков. Трансляция даже короткой программы из нескольких строк могла длиться минутами.

Основными вариантами работы на компьютере были монопольный или пакетный режим. В первом случае компьютер целиком передавался в распоряжение программиста на определенное (как правило,

небольшое) время[5]. За это время в обстановке спешки (машину надо освобождать) удавалось несколько раз оттранслировать и запустить программу, но возможности спокойно проанализировать ошибки и результаты не было никакой. Машина в этом случае использовалась не слишком эффективно, поскольку простаивала большую часть времени, пока исправлялись ошибки и готовился очередной «прогон».

При работе в пакетном режиме программист обычно вообще не видел компьютера. Написанные на бланках программы передавались на перфорацию, а затем колоды перфокарт — на компиляцию и счет. Может быть, машины в этом случае были загружены и лучше, но даже в хорошо организованных вычислительных центрах[6] не удавалось запускать программу чаще 2–3 раз в день.

Все эти неудобства привели к созданию многопользовательских систем, использующих режим разделения времени. В рамках одного из таких проектов и возник язык Бейсик, который можно было использовать в интерактивном режиме в системе с разделением времени.

Бейсик

Система с разделением времени представляет собой достаточно мощный компьютер, к которому подключены терминалы. Центральная машина обслуживает терминалы по очереди, уделяя каждому небольшой квант времени, измеряемый миллисекундами. Поскольку цикл обслуживания терминалов достаточно мал, у каждого пользователя создается иллюзия, что он монопольно подключен к компьютеру. В качестве терминалов в таких системах первоначально использовались телетайпы.

Такая интерактивная система на базе компьютера General Electric была развернута в 1964 году в Дартмутском колледже (США) благодаря усилиям руководителя факультета математики этого колледжа Джона Кемени (John Kemeny) и директора вычислительного центра Томаса Курца (Thomas Kurtz). Они же реализовали и новый язык для этой системы, который назвали BASIC (Бейсик).

[5] Поскольку компьютеров было мало, а желающих ими воспользоваться — много, то работа шла круглосуточно. Немалой удачей считалось заполучить машину на целую ночь. В начале 1980-х мне посчастливилось участвовать в таких бдениях.

[6] Например, в вычислительном центре Ленинградского политехнического института.

Официальная расшифровка названия языка — Beginner's All-purpose Symbolic Instruction Code — многоцелевой код (язык) символических команд для начинающих, — довольно неуклюжа. Однако вполне очевидно, что авторам было важно само слово «basic», означающее по-английски «базисный, основной». Кроме того, в англоговорящей среде хорошо известен «Basic English» — упрощенный вариант английского языка, предложенный в 1930 году лингвистом Чарльзом Огденом (Charles Ogden). Огден отобрал 850 слов, с помощью которых, по его мнению, можно объяснить значение всех других. Примечательно, что среди этих слов было всего 14 глаголов. Грамматика также была предельно упрощена. Basic English предназначался для первоначального освоения английского как второго языка, и продвигался в качестве международного языка. Выбирая название языка для начинающих, Кемени и Курц, наверняка, имели в виду аналогию с Basic English.

Первые две программы на Бейсике была запущены совместно в системе с разделением времени Дартмутского колледжа 1 мая 1964 года.

Первоначальная версия Бейсика была очень проста. Предусматривалось 14 видов операторов (а в Basic English 14 глаголов!) и возможность работы только с вещественными числами. В течение 1964–1971 годов авторами было внесено в язык много дополнений: средства работы с матрицами, строками, файлами, форматный вывод. В Дартмутских реализациях Бейсика использовались компиляторы.

Вскоре на Бейсик обратили внимание компании, производящие вычислительную технику. Бейсик стали включать в состав программного обеспечения своих мини-компьютеров фирмы Hewlett-Packard, Digital Equipment (DEC) и другие.

По-настоящему массовым языком Бейсик стал с появлением персональных компьютеров в конце 1970-х — начале 1980-х годов. Практически все модели первых ПК оснащались интерпретатором Бейсика, который иногда даже записывался в постоянное запоминающее устройство машины и был готов к работе сразу после включения компьютера. Иногда Бейсик оказывался единственным приемлемым средством программирования на таких компьютерах. Тогда же Бейсик получил широкое распространение в системе школьного образования, куда он попал вместе с персональными компьютерами.

Каждая модель компьютера имела свой диалект Бейсика. Разработчики непременно вносили в язык изменения, обусловленные как особенностями машин, так и желанием расширить возможности языка. Большинство реализаций Бейсика на ПК включали средства для работы с графикой и звуком. Хотя некоторое ядро языка все же сохранялось, но с тех пор говорить о Бейсике, как о едином языке, довольно трудно. Эта ситуация изрядно угнетала авторов языка, переживавших за свое детище.

Были предприняты попытки стандартизации Бейсика. В 1978 году принят ANSI-стандарт «минимального Бейсика», а в 1988, при непосредственном участии Дж. Кемени и Т. Курца — «стандартного Бейсика». Однако эти стандарты не оказали заметного влияния на развитие языка, которое оставалось стихийным.

Основные черты Бейсика

Здесь будут названы свойства некоторой усредненной версии, которые присутствовали в большинстве диалектов, бывших в употреблении в 1980-е и в начале 1990-х годов, когда применение языка, особенно в системе образования, было по-настоящему массовым.

- Отсутствие структуры. Программа — это набор пронумерованных строк. Номера используются в качестве меток и для редактирования программы.
- Все переменные — глобальные.
- Предусмотрены подпрограммы без параметров и однострочные функции, определяемые пользователем.
- Вещественный и строковый типы данных, иногда добавляется целый тип.
- Управляющие операторы: `IF` (иногда без `ELSE` и ограниченный одной строкой), `GOTO`, `FOR`, иногда `WHILE`.
- Динамические массивы переменного размера.
- Средства для интерактивной работы: ввод с терминала, вывод на экран.
- Файловый ввод-вывод.
- Средства для считывания и записи фрагментов программы (благодаря этому может быть получена самомодифицирующаяся программа).

- Интеграция в язык функций операционной системы: загрузка и запуск программ, получение оглавления диска, удаление и переименование файлов и т. п.

Благодаря реализации с помощью интерпретатора Бейсик-системы имели неплохие возможности отладки, такие, как трассировка хода выполнения программы, возможность остановки программы и продолжения ее работы с того же места, просмотр и изменение значений переменных. Компилирующие системы для «больших» языков такими возможностями в те годы не обладали.

Главным же достоинством была работа в интерактивном режиме. После мытарств пакетной обработки непосредственное взаимодействие с программой в режиме диалога, возможность запустить ее столько раз, сколько необходимо, воспринимались как настоящий прорыв в организации программистского труда.

Примеры программ на Бейсике

Легкость подготовки и запуска программ вроде «Hello, World!» — это ключевое достоинство Бейсика. Не зная ничего ни про компиляторы, ни про операционную систему, можно ввести в командной строке Бейсик-интерпретатора:

```
PRINT "Hello, World!"
```

и тут же получить в ответ:

```
Hello, World!
Ok
```

Это, как говорят, непосредственный режим исполнения команды. Если команде предшествует номер, то в соответствии с этим номером она вставляется в программу, а не выполняется немедленно.

```
10 PRINT "Hello, World!"
LIST
10 PRINT "Hello, World!"
Ok
RUN
Hello, World!
Ok
```

Курсивом отмечен вывод интерпретатора.

Теперь сортируем вставками массив случайных чисел. Программа приведена целиком, включая заполнение и вывод массива (листинг 1.6).

Листинг 1.6. Сортировка вставками на Бейсике

```
10 REM Программа сортировки массива
20 PRINT "Сортировка случайных чисел"
30 INPUT "Число элементов "; N
40 DIM A(N)
45 FOR I = 1 TO N
50    A(I) = INT(100 * RND): PRINT A(I);
60 NEXT I
70 GOSUB 1000: REM на подпрограмму сортировки
75 PRINT : PRINT
80 FOR I = 1 TO N
85    PRINT A(I);
90 NEXT I
93 PRINT
95 END
1000 REM подпрограмма сортировки
1010 FOR I = 2 TO N
1020    X = A(I): A(0) = X: J = I – 1
1030    IF X >= A(J) GOTO 1060
1040       A(J + 1) = A(J): J = J - 1
1050    GOTO 1030
1060    A(J + 1) = X
1070 NEXT I
1080 RETURN
```

Обратите внимание, что массив создается динамически после ввода данных о его размере. Массив, созданный оператором `DIM A(N)`, содержит `N+1` элементов, индексы которых начинаются с нуля. Я использовал нулевой элемент в роли барьера для внутреннего цикла, помещая в `A(0)` значение `I`-го элемента.

Точки с запятой в конце некоторых операторов `PRINT` существенны. Они позволяют блокировать перевод строки после печати. На Паскале в такой ситуации применяли бы `Write` вместо `Writeln`.

Использование подпрограммы в этом примере неоправданно. Однако разделение программы на части полезно само по себе, к тому же появляется повод записать дополнительный комментарий. Но поскольку переменные — глобальные, составляя и используя эту подпрограмму, все время приходится помнить об опасности совпадения имен. Не зная, какие вспомогательные переменные употреблены в подпрограмме, нельзя писать и остальные части программы. К тому же подпрограмма сортирует массив только с названием `A`, размер ко-

торого должен обозначаться непременно `N`. Я давно не пользовался языком, не предусматривающим локальных имен, и испытал заметный дискомфорт при подготовке этого примера.

Использование переменной `I` в роли счетчика цикла и в основной программе и в подпрограмме — прием весьма рискованный, хотя в этом конкретном случае к неприятностям и не приводит.

Бейсик от Microsoft

Легенда гласит, что интерпретатор Бейсика для компьютера Altair, разработанный в 1975 году Биллом Гейтсом (Bill Gates) вместе со своим школьным товарищем Полом Алленом (Paul Allen), стал первым продуктом компании Microsoft. С тех пор Microsoft уделяет Бейсику немалое внимание.

Вначале вместе с операционной системой MS DOS Microsoft поставляла обладавшие скромными возможностями интерпретаторы Бейсика (BASICA, GW-BASIC), которые реально могли использоваться лишь для решения ученических задач и написания несложных прикладных программ. Но затем в конце 80-х была выпущена система программирования QuickBasic. Ее упрощенный вариант QBasic входил в комплект MS DOS. Система имела интегрированную среду разработки с полноценным редактором программ. Входной язык был значительно расширен по сравнению с существовавшими до этого диалектами Бейсика. В то же время из языка был исключен ряд команд, предназначавшихся для работы с текстом программы: `RUN`, `LIST` и т. п. Необходимость в них пропала в связи с появлением интегрированной среды. В состав системы входил компилятор. Система стала пригодна для разработки больших программ.

В 1991 году был выпущен новый продукт Microsoft — Visual Basic. Это была первая среда визуального программирования для Windows — операционной системы с графическим пользовательским интерфейсом, начинавшей приобретать тогда все большую популярность.

Разработка прикладных программ для Windows с помощью традиционных систем программирования была трудным делом. Требовались знание непростой организации Windows и масса усилий для создания графического пользовательского интерфейса. Система Visual Basic (рис. 1.8) избавляла программиста от этих проблем. Интерфейс стало возможным просто рисовать, пользуясь имеющимися инстру-

ментами. А от разработки программы целиком программист был освобожден. Его часть работы состояла теперь лишь в написании процедур обработки событий для тех элементов, которые он выбрал при создании пользовательского интерфейса (окна, меню, кнопки, списки и т. д.).

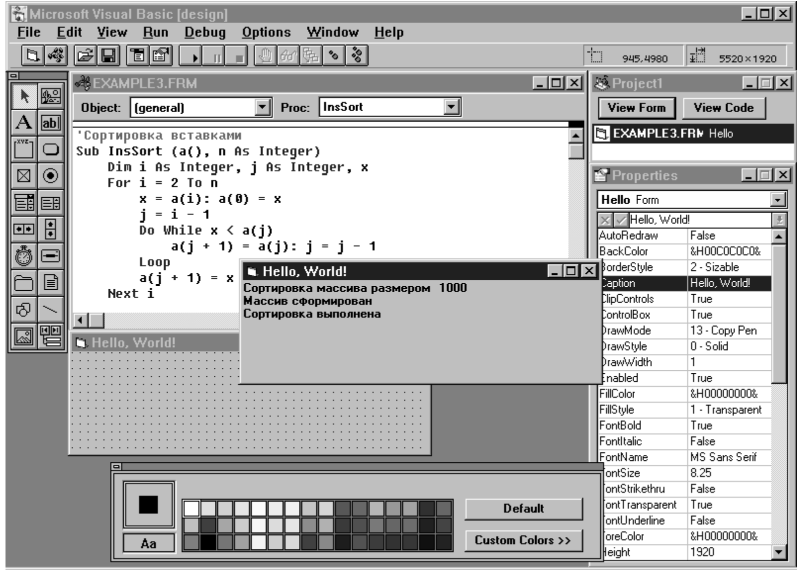


Рис. 1.8. Visual Basic 3.0

Входной язык Visual Basic незначительно отличался от языка QuickBasic. Язык и интегрированная среда стали практически неотделимы, а написание программы вне среды потеряло смысл. Функциями среды стали контроль над процессом структурирования программы, автоматическое формирование (и форматирование) элементов программы.

Ниже приводится перечень основных свойств языка Visual Basic 6.0, последней версии, выпущенной до перехода на появившуюся в 2000 году платформу Microsoft .NET.

- Необязательность (и ненужность) нумерации строк. По желанию программиста можно установить обязательность описаний.

- Совместимость с предшествующими версиями Бейсика. Возможно написание фрагментов программы «в старом стиле»: с номерами строк, переходами (GOTO и GOSUB) и т. п.
- Более 70 разновидностей операторов. По старой традиции Бейсика многие средства (такие как ввод-вывод), которые в других языках реализуются библиотеками, здесь являются частью языка. Это усложняет работу, поскольку каждый оператор имеет уникальный синтаксис, в случае же реализации с помощью библиотек синтаксис един — вызов процедуры или функции.
- Процедуры (Sub – End Sub) и функции (Function – End Function) с параметрами и локальными переменными.
- Типы данных: Byte, Boolean, Integer, Long — длинные целые, Single — короткие вещественные, Double — вещественные двойной точности, Currency — денежные единицы, Decimal — двадцатидевятизначные десятичные числа с фиксированной точкой, Date — календарная дата, Object — указатели на объекты, String — строки, Variant — переменная этого типа (Variant — тип по умолчанию) может содержать любые данные. При операциях с переменными типа Variant выполняются неявные преобразования типов.
- Массивы: статические (объявляются с помощью оператора Dim) и динамические (ReDim).
- Записи (структуры), называемые «пользовательский тип данных».
- Элементы объектно-ориентированного стиля программирования. Классы — пользовательский тип данных с методами (процедурами и функциями) и свойствами, но без наследования.
- Частичный контроль соответствия типов.
- Структурные управляющие операторы:
```
If - Then - ElseIf - Else - End If
Select - Case - End Select
Do - While - Until - Exit Do - Loop
Do - Exit Do - Loop - While - Until
For Each - In - Exit For - Next
For - To - Step - Exit For - Next
```
- Три разновидности модулей, не имеющих специального синтаксического оформления: модули форм — набор процедур обработки событий; стандартные модули — содержат общие части программы, не относящиеся к конкретному элементу экранной формы; мо-

дули классов — содержат определения пользовательских типов данных (классов).

- Реализации первых версий Visual Basic основывались на компиляции программы в промежуточное представление (P-код[7]) и последующей интерпретации P-кода. Это обеспечивало быструю трансляцию и компактность получаемых файлов, но снижало быстродействие. Начиная с версии 5.0, Visual Basic оснащается компилятором в машинный код. Возможность использования P-кода также сохранена.

На платформе Microsoft .NET Бейсик продолжает оставаться одним из основных языков. Но версия Visual Basic для .NET несовместима с предшествующими. Старые программы не исполняются в новой среде.

Пример программы на Visual Basic

Чтобы не загромождать свой рассказ двумя длинными листингами, я совместил два традиционных примера в одном. Окно нашей программы (Visual Basic для того и нужен, чтоб создавать оконные приложения) будет называться «Hello, World!», а выполнять программа будет сортировку массива случайных чисел методом простых вставок. Окно программы можно увидеть на рисунке 1.8, где оно показано в среде Visual Basic 3.0. Чтобы не усложнять программу и сохранить ее текст обозримым, я не использовал никаких диалоговых элементов. По щелчку мыши в пределах окна программа формирует массив и начинает его сортировку, выдав сообщение. По завершении сортировки также печатается сообщение. Текст программы, точнее, модуля формы, показан в листинге 1.7. Я отладил программу с помощью Visual Basic 1.0 для DOS, а затем убедился в ее полной совместимости с версиями 3.0 и 6.0. Приведенный текст сохранен из среды Visual Basic 3.0, в этом случае он оказался короче всего. Фрагменты, созданные автоматически, выделены в листинге курсивом. Visual Basic отформатировал программу, записав служебные слова в принятом для этой среды стиле, а в качестве версии языка в первой строке указал 2.0.

---

[7] P-код (произносится «Пи-код») — машинный код некой несуществующей (виртуальной) машины, как правило, со стековой организацией. Структура P-кода удобна как для компиляции в этот код исходной программы, так и для последующей интерпретации. Впервые P-код использован в одной из первых реализаций языка Паскаль (отсюда и буква P в названии). Подобный принцип лежит в основе реализаций языка Ява.

Листинг 1.7. «Hello, World!» и сортировка на Visual Basic

```
VERSION 2.00
Begin Form Hello
   BackColor       =   &H00C0C0C0&
   Caption         =   "Hello, World!"
   ClientHeight    =   1515
   ClientLeft      =   1005
   ClientTop       =   5325
   ClientWidth     =   5400
   ForeColor       =   &H00000000&
   Height          =   1920
   Left            =   945
   ScaleHeight     =   1515
   ScaleWidth      =   5400
   Top             =   4980
   Width           =   5520
End
Option Explicit 'обязательность описаний

Sub Form_Click ()
   Const n = 1000
   Dim i As Integer
   ReDim a(n)
   Print "Сортировка массива размером "; n
   For i = 1 To n
      a(i) = Int(100 * Rnd)
   Next i
   Print "Массив сформирован"
   InsSort a(), n
   Print "Сортировка выполнена"
End Sub

'Сортировка вставками
Sub InsSort (a(), n As Integer)
   Dim i As Integer, j As Integer, x
   For i = 2 To n
      x = a(i): a(0) = x
      j = i - 1
      Do While x < a(j)
         a(j + 1) = a(j): j = j - 1
      Loop
      a(j + 1) = x
   Next i
End Sub
```

Оператор `Option Explicit` устанавливает обязательность описаний, которые выполнены с помощью операторов `Dim`, а для динамического массива — с помощью `ReDim`. Привычный для Бейсика `Print` здесь имеет несколько иной статус. Это метод объекта `Form`, и полная запись могла бы выглядеть как `Hello.Print`, где `Hello` — название формы (см. вторую строку листинга).

Обратите внимание что, если исключить из приведенного листинга все, что связано с сортировкой, то получится «Hello, World!» в чистом виде. Ни одной строчки на Бейсике для получения такой программы писать вообще не нужно.

Популярность Бейсика

С самого создания основными достоинствами Бейсика были диалоговый режим работы и низкий барьер вхождения. Первое делало Бейсик-системы привлекательными для тех, кто до этого работал в пакетном режиме; второе — для новичков и непрофессиональных программистов — специалистов прикладных областей, которым было необходимо выполнять расчеты, не связанные с созданием больших программ. Бейсик был очень популярен в сфере образования, поскольку позволял научить составлению простых программ за считанные часы.

Между тем эффект быстрого вхождения имел и оборотную сторону. Возможность быстро начать объяснялась тем, что язык, по существу, не имел структуры, не требовал описаний, а значит, ненужным было и рассмотрение какой-либо системы типов и других предварительных соглашений. Если для программ из нескольких десятков или даже сотен строк такое примитивное устройство не было помехой, то с ростом размера программы трудности резко возрастали. Те, кому было достаточным работать с небольшими программами, могли быть вполне довольны. Опытные программисты, владевшие более развитыми языками, могли справляться с неструктурированностью Бейсика, моделируя его средствами возможности «настоящих» языков. Но больше всего пострадали те, для кого язык, казалось бы, и предназначался, — начинающие программисты, те, для кого Бейсик оказывался первым языком. Приучившись с самого начала мыслить категориями Бейсика и не будучи знакомыми до поры ни с какими альтернативами, они чаще всего приобретали не только неправильный стиль программирования, но и порочный стиль мышления.

Известно высказывание одного из авторитетнейших в мире специалистов в области программирования Эдсгера Дейкстры (Edsger Wybe Dijkstra), основоположника структурного программирования, изобретателя ряда фундаментальных алгоритмов: «Практически невозможно научить хорошему программированию студентов, ориентированных первоначально на Бейсик: как потенциальные программисты они умственно оболванены без надежды на исцеление». Сказано в 1982 году. Наблюдения вполне подтверждают этот тезис, хотя надежда, думаю, все-таки остается. Отличить молодого человека, «овладевшего» в детстве Бейсиком, можно почти безошибочно. Даже когда он начинает писать программы, к примеру, на Паскале, то все равно в его программе все переменные — глобальные, все процедуры — без параметров, все циклы — `for`.

С появлением Quick/Visual Basic оценка ситуации меняется. Этот диалект, а по существу — другой язык, содержит многие элементы, необходимые для разработки больших программных комплексов. При определенной самодисциплине можно создавать прикладные системы приемлемого качества. Тем более что мощная и довольно легкая в освоении интегрированная среда разработки берет на себя многие контрольные функции, помогающие сохранить целостность программы и обеспечить определенный уровень контроля за взаимным соответствием модулей.

В обновленном виде Бейсик по-прежнему занимает нишу достаточно простых в освоении интерактивных средств разработки прикладных программ.

Microsoft продвигает Бейсик и еще в одной роли. Он стал единым внутренним языком программирования (средством создания макрокоманд, макросов) офисных программ Microsoft: электронной таблицы Excel, текстового процессора Word, системы управления базами данных Access. Эта версия называется Visual Basic for Application — Visual Basic для приложений, сокращенно VBA.

Как язык программирования, Visual Basic обладает рядом недостатков. Он отягощен наследием прошлого, что усложняет язык и провоцирует смешение стилей. Сохранение совместимости с прежними версиями приводило к тому, что синтаксис плохо защищал программиста от ошибок. Неполон контроль соответствия типов, а появление типа Variant даже усиливает возможности использования неявных преобразований.

Сравнивать язык Visual Basic с такими языками как Фортран, Си, Ява, Ада, Оберон не вполне правомерно. Visual Basic, в отличие от названных языков, это язык одной операционной системы и одной фирмы.

В нашей стране до появления персональных компьютеров Бейсик оставался почти не замеченным. Начиная с 1969 года, были осуществлены ряд реализаций (Горьковский университет, Институт математики Академии наук Белоруссии, Вычислительный центр Сибирского отделения Академии наук, Институт космических исследований), но заметного распространения они не получили. С появлением мини-ЭВМ в 70-е годы известность Бейсика несколько возросла. Эти машины (М-6000, М-7000, СМ-1, СМ-2) были клонами мини-компьютеров Hewlett Packard и DEC (СМ-3, СМ-4), а в составе их математического обеспечения (тогда говорили именно «математического») имелись системы программирования на Бейсике (американские). Я тоже тогда впервые услышал про Бейсик, имевшийся на М-6000. Работать на Бейсике, как мне рассказывали, было очень удобно.

Картина резко изменилась с появлением первых микрокомпьютеров, которые у нас еще не назывались персональными, да и компьютерами тоже не назывались — ЭВМ. Бейсик на этих машинах был зачастую единственной системой, пригодной для решения прикладных задач.

С 1985 года в средней школе был введен предмет «Основы информатики и вычислительной техники». Начался процесс хоть и медленного, но организованного проникновения компьютеров в учебные заведения. И, несмотря на усилия, предпринятые для внедрения правильного подхода к обучению на основе хорошо структурированного языка (А. П. Ершов и др.), очень многие преподаватели вместе с учениками принялись осваивать в первую очередь Бейсик. О последствиях такого подхода незадолго до этого предупреждал Э. Дейкстра.

Однако, как мне представляется, несмотря на очевидный интерес к языку, в те годы не было сколько-нибудь крупных проектов, где бы использовался Бейсик. Разрабатывались простые игровые и учебные программы, размер которых редко превышал тысячу строк.

Новая версия QuickBasic от Microsoft практически не была замечена отечественными разработчиками. Известны лишь отдельные примеры использования QuickBasic в серьезных проектах.

Не обратили внимания у нас и на первые версии Visual Basic. В начале 1990-х годов (Visual Basic появился в 1991 г.), когда массовое распространение IBM PC-совместимых компьютеров только началось, превалировали устаревшие модели машин, которые были попросту непригодны для использования Windows, и спроса на программы для Windows не было. Когда спрос сформировался, появилась система программирования Delphi (в 1995 году), которая обладала такими же возможностями визуального программирования, но была основана на языке Паскаль, который в среде отечественных программистов был к этому времени очень популярен. В конце 1990-х ситуация меняется. В связи с широким распространением технологий Microsoft в целом возрос интерес и к Visual Basic. На рубеже веков в нашей стране Delphi заметно превосходил Visual Basic по популярности, в мире в целом внимание к этим двум системам было примерно одинаково.

Структурное программирование

У меня уже неоднократно был повод упомянуть структурное программирование. Настал момент обсудить это явление по существу. Напомню, что в разговоре о генеалогии языков мы как раз подошли к концу 1960-х годов, когда появилась эта технология.

К тому времени в мире был осуществлен ряд крупных программных разработок, в ходе которых проявилось несовершенство существовавшей технологии программирования. Программы страдали от большого числа ошибок, на поиск и устранение которых тратилось много времени и ресурсов. Установленные сроки разработок постоянно срывались, а запланированные бюджеты многократно перекрывались. Производительность труда программистов оказывалась низкой и оценивалась в 5–10 отлаженных операторов на человека в день. Это, конечно, не означает, что программисты в те времена, написав несколько строчек за день, отдыхали. Счет ведется по готовой программе. Такая цифра получалась, если разделить число команд в готовой и отлаженной программе на время ее разработки и на число участников разработки.

Одним из крупнейших программных проектов того времени была разработка операционной системы для компьютеров серии IBM/360[8].

[8] В нашей стране эта операционная система эксплуатировалась под названием ОС ЕС.

При создании этой системы в полной мере проявили себя все перечисленные проблемы. В ее создании участвовало более 1000 человек одновременно, а трудоемкость составила 5000 человеко-лет. Сроки и бюджет были значительно превышены. При этом, по оценкам, в уже готовой системе оставалось как минимум 1000 ошибок.

Сложившуюся ситуацию даже называют кризисом программирования 60-х годов. Выходом из этого кризиса послужило структурное программирование. Это понятие связывают в первую очередь с именем профессора Эйндховенского технического университета (Голландия) Эдсгера Дейкстры (1930–2002).

В мартовском номере журнала «Communications of the ACM»[9] за 1968 год он опубликовал небольшую статью «О вреде оператора go to» («Go To Statement Considered Harmful»). В ней говорилось, что качество программы есть убывающая функция количества использованных в ней операторов перехода. Дейкстра предлагал отказаться от использования переходов, а для организации управления в программе использовать только конструкции ветвлений **`if`** – **`then`**, **`if`** – **`then`** – **`else`**, **`case`** и циклов, подобных **`while`** и **`repeat`**. В этом случае, как указывал Дейкстра, статический текст программы будет наиболее простым образом соответствовать динамическому процессу ее выполнения, что дает больше шансов программисту контролировать ход вычислений, порождаемых его программой.

В своей статье Дейкстра отметил, что мысль о нежелательности использования **`goto`** высказывалась и раньше, в частности Ч. Э. Р. Хоаром и Н. Виртом в статье, опубликованной в CACM в 1966 году, где речь шла об их проекте развития Алгола-60. Позже, в 1972 году Дейкстра опубликовал работу «Заметки по структурному программированию», в которой развил свою концепцию и показал, что кроме прочего, для структурированной программы облегчается доказательство ее правильности.

Структурное программирование — не самоцель, хотя введение определенной дисциплины программирования («Дисциплина программирования» — название еще одной работы Э. Дейкстры) и стандартизация стиля хороши уже сами по себе, особенно если речь идет

---

[9] Communications of the ACM, сокращенно CACM — журнал Ассоциации по вычислительной технике (Association for Computing Machinery, ACM), одной из авторитетнейших и старейших (основана в 1947 году) организаций в области обработки информации.

о создании больших программных комплексов крупными коллективами программистов. Упорядочение управляющей структуры программы упрощает ее восприятие, облегчает чтение (говорят даже о «читабельности» программ) и понимание как коллегами-программистами, так и самим автором программы. А это, в свою очередь, снижает вероятность появления ошибок и облегчает их поиск, увеличивая тем самым надежность программ.

При всей своей простоте оператор перехода `goto` — очень мощное средство. Из любой точки программы можно передать управление почти в любую другую точку. Неопытный программист, не ощущая опасности таких переходов, может очень запутать ими организацию управления в программе. И если тем современным программистам, кто учился по хорошим книгам, с хорошим преподавателем или на хороших примерах, такая опасность почти не грозит, то в прошлом ситуация была совсем иной. Даже на обложке журнала «Программирование» долгие годы красовался алгоритм Евклида, записанный с помощью `goto`. Для программирования с беспорядочными переходами придумали название «стиль спагетти». Если показать стрелками направление передач управления в такой программе, то картина будет напоминать тарелку длинных итальянских макарон. Листинг 1.8 демонстрирует реальную программу, написанную специалистом по системам водоочистки, умным, интеллигентным человеком, но неопытным программистом. Количество переходов в ней впечатляет. Стрелки вы можете дорисовать сами.

Листинг 1.8. Программирование в стиле спагетти

```
      PROGRAM
      READ(5)A,C,P
      IF (A.LE.20) GO TO 1
   12 IF (A.LE.30) GO TO 2
   16 IF (A.LE.120) GO TO 3
   20 IF (A.LE.300) GO TO 4
   24 IF (A.LE.1000) GO TO 5
   27 IF ((A-50).LE.1450) GO TO 6
   31 IF (A.LE.1500) GO TO 7
   35 IF (A.LE.1500) GO TO 8
   39 IF (A.LE.1500) GO TO 9
   43 IF (A.LE.1500) GOTO 10
      WRITE(7,11)
   11 FORMAT('УСЛОВИЯ НЕ СООТВЕТСТВУЮТ СНИП')
      GO TO 50
```

```
 1 IF (C.LE.50) GO TO 13
   GO TO 12
13 IF (P.LE.50000) GO TO 14
   GO TO 12
14 WRITE(7,15)
15 FORMAT('СКОРЫЕ ОДНОСТУПЕНЧАТЫЕ ОТКРЫТЫЕ ФИЛЬТРЫ')
   GO TO 50
 2 IF (C.LT.50) GO TO 17
   GO TO 16
17 IF (P.LE.5000) GO TO 18
   GO TO 16
18 WRITE(7,19)
19 FORMAT('СКОРЫЕ НАПОРНЫЕ ФИЛЬТРЫ')
   GO TO 50
 3 IF (C.LE.120) GO TO 21
   GO TO 20
21 WRITE(7,22)
22 FORMAT('КОНТАКТНЫЕ ОСВЕТЛИТЕЛИ')
   GO TO 50
 4 IF (C.LE.120) GO TO 23
   GO TO 24
23 WRITE(7,25)
25 FORMAT('КОНТАКТНЫЕ ПРЕФИЛЬТРЫ')
   GO TO 50
 5 IF (C.LE.120) GO TO 26
   GO TO 27
26 IF (P.LE.800) GO TO 28
   GO TO 27
28 WRITE(7, 29)
29 FORMAT('ТРУБЧАТЫЕ ОТСТОЙНИКИ И НАПОРНЫЕ ФИЛЬТРЫ')
   GO TO 50
 6 IF (C.LE.120) GO TO 30
   GO TO 31
30 IF (P.LE.5000) GO TO 32
   GO TO 31
32 WRITE(7,23)
33 FORMAT('ОСВЕТЛ. СО ВЗВЕШ. ОСАДКОМ - СКОРЫЕ ФИЛЬТРЫ')
   GO TO 50
 7 IF (C.LE.120) GO TO 34
   GO TO 35
34 IF (P.LE.30000) GO TO 36
   GO TO 35
36 WRITE(7,37)
37 FORMAT('ГОРИЗОНТАЛЬНЫЕ ОТСТОЙНИКИ - СКОРЫЕ ФИЛЬТРЫ')
```

```
      GO TO 50
    8 IF (C.LE.120) GO TO 38
      GO TO 39
   38 IF (P.LE.5000) GO TO 40
      GO TO 39
   40 WRITE(7,41)
   41 FORMAT('ВЕРТИКАЛЬНЫЕ ОТСТОЙНИКИ - СКОРЫЕ ФИЛЬТРЫ')
      GO TO 50
    9 IF (C.LE.120) GO TO 42
      GO TO 43
   42 WRITE(7,44)
   44 FORMAT('ДВЕ СТУПЕНИ ОТСТОЙНИКОВ - СКОРЫЕ ФИЛЬТРЫ')
      GO TO 50
   10 IF (C.LE.50) GO TO 45
      GO TO 50
   45 WRITE(7,46)
   46 FORMAT('МЕДЛ. ФИЛ. С МЕХ. ИЛИ ГИДР. РЕГЕНЕР. ПЕСКА')
   50 STOP
      END
```

Справедливости ради надо сказать, что задача (выбор типа фильтра) довольно специфична, так что такая организация программы в некоторой мере оправдана. К тому же писать приходилось, как видите, на примитивном диалекте Фортрана.

Основы структурного программирования

Итак, в первом приближении, структурное программирование — это программирование без `goto`. В языках, где есть развитые управляющие операторы, без переходов обойтись, пожалуй, можно. Но есть языки, где без переходов написать программу просто нельзя. Например, старые версии Фортрана и Бейсика, язык ассемблера. Можно ли говорить о структурном программировании на таких языках и помогает ли структурное программирование бороться со сложностью задач в этом случае? Попробуем ответить на этот вопрос и сформулируем, что же следует называть структурным программированием.

Структурное программирование — это дисциплина, требующая, чтобы любая программа строилась из ограниченного набора *типовых (базовых) управляющих структур* и их *композиций*.

Что же следует считать базовыми управляющими структурами? Чтобы не быть привязанными к какому-то конкретному языку, воспользуемся блок-схемами. На рисунке 1.9 показан один из возможных

наборов. Этот набор соответствует операторам, которые есть во многих развитых языках программирования. В скобках записаны служебные слова, которые обычно оформляют соответствующую конструкцию. Только схема «цепочка», выражающая последовательное (линейное) выполнение двух действий, не имеет такого обозначения.

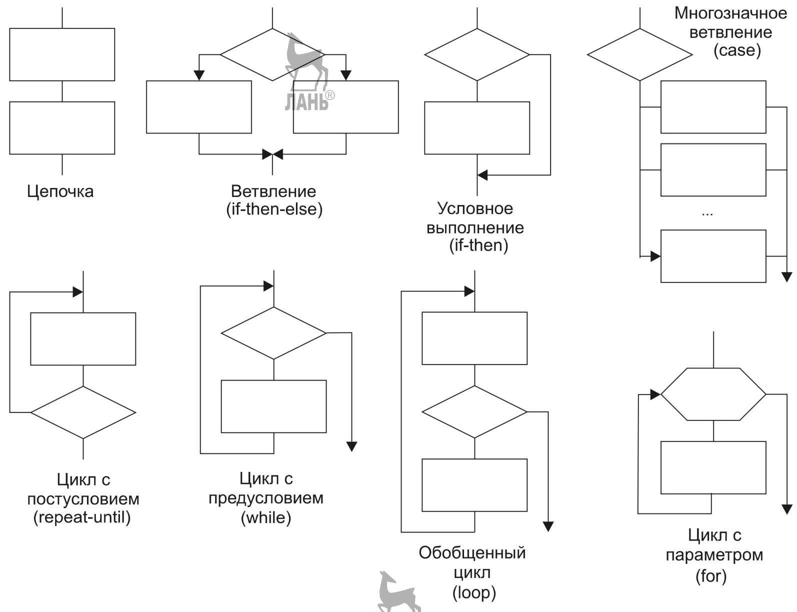


Рис. 1.9. Базовые управляющие структуры

Прямоугольный блок на схеме (назовем его функциональным) изображает *действия*. Не важно, какие именно, важно, что эти действия начинаются в какой-то момент и в какой-то момент заканчиваются. Соответственно, у этого блока один вход и один выход. Действия могут быть простыми или, наоборот, очень сложными.

Ромбы — это блоки *проверки условий*. Каждое условие может выполняться или нет (быть истинным или ложным). У блока проверки один вход и два выхода. На один выход попадаем, если условие выполнено, на другой — если нет. Пока не важно, в каком случае какой из двух выходов используется.

На схеме многозначного ветвления ромб исполняет несколько иную роль. Он изображает проверку некоторого значения (предполагается, что оно может быть обозначено внутри ромба), а боковые ответвления выходной линии потока ведут к действиям, которые выполняются, когда значение совпадает с одной из заданных констант.

На схеме цикла с параметром использован шестиугольный блок. Верхний его вход означает начало всей последовательности действий, а боковой — переход к следующему значению параметра. Нижний выход из блока ведет к выполнению цикла, когда выбрано очередное значение параметра, правый выход — завершение цикла из-за исчерпания значений параметра.

Внутри блоков не обозначены никакие конкретные действия. Они нас пока не интересуют. Такие схемы без конкретных обозначений как раз и называют «управляющие структуры». Очень важно, что все показанные структуры имеют ровно один вход и ровно один выход. А, значит, нет никаких препятствий, чтобы вкладывать одну структуру внутрь другой. Любая из этих структур может представлять внутренность одного из функциональных блоков другой структуры. Образуются композиции базовых структур. На рисунке 1.10 показаны примеры таких композиций. Отдельно нужно отметить необходимость включения в рассматриваемый набор структуры «цепочка». Без нее было бы невозможно предусмотреть выполнение одного действия вслед за другим.

Таким образом, даже на языке, где нет достаточного набора структурных операторов, можно программировать в структурном стиле, используя условные и безусловные переходы для организации управления по приведенным схемам.

В связи с рассмотрением базового набора управляющих структур возникают как минимум три вопроса:

1. Достаточен ли такой набор для записи программы? Интуитивно представляется что да, достаточен. Большинство читателей, я думаю, программируют, не используя `goto`, или даже на языках, где этого оператора нет. Но, может быть, существует такая программа (блок-схема), которую с помощью такого набора не представить?
2. Если ответ на первый вопрос утвердительный, то есть рассматриваемый набор достаточен, то нельзя ли обойтись меньшим набором?

3. Если ответы на первые два вопроса утвердительны, то каков минимальный набор?

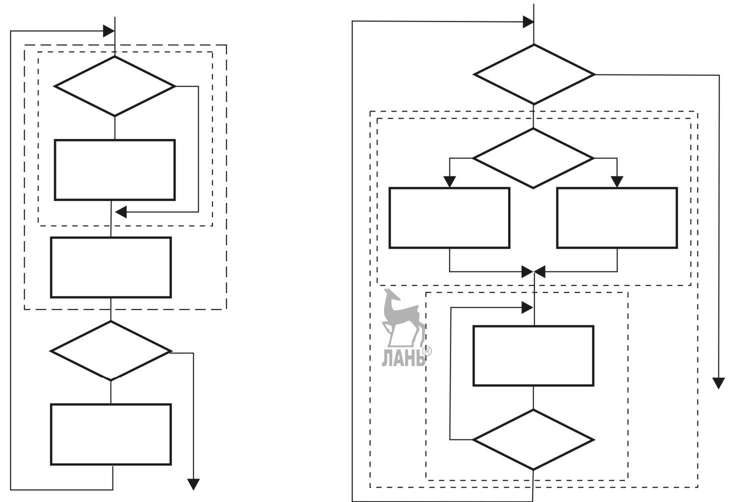


Рис. 1.10. Композиции управляющих структур

Ответы дает теорема структурирования. На первые два вопроса ответ положительный. А на третий вопрос, читатель, попробуйте ответить сами до того, как мы эту теорему рассмотрим.

Вполне очевидно, что можно обойтись без цикла с параметром, который выражается через другие структуры. Многозначное ветвление легко программируется с помощью `if` – `then` – `else`. Отбрасывая эти структуры, мы можем рассматривать только схемы, где используются функциональные блоки (прямоугольники) и блоки проверки условий (ромбы) с двумя выходами. Обобщенный цикл — на то и обобщенный, что он может заменить циклы с пред- и постусловием. Конструкция `if` – `then` — это частный случай `if` – `then` – `else`. Остаются три структуры. А можно ли меньше?

Пример неструктурированной программы

Возьмем несложную задачу, на примере которой будем обсуждать вопросы структурирования.

Задача. Значение наибольшего из трех чисел *a*, *b, c* присвоить переменной *M*.

В первом варианте решения будем использовать для определения максимума только парные сравнения исходных чисел. Блок-схема решения показана на рисунке 1.11. Такое решение позволяет всегда получить ответ с помощью ровно двух сравнений и одного присваивания.

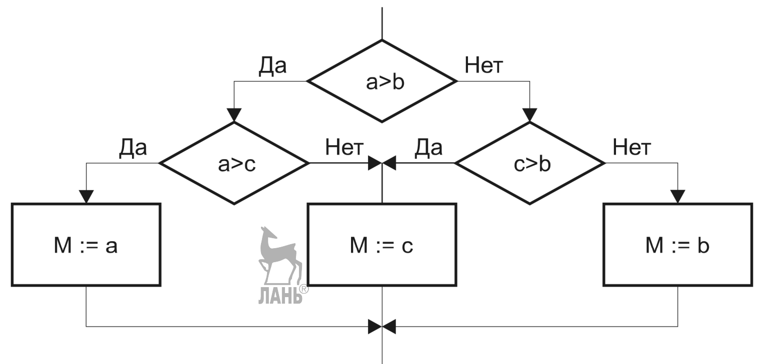


Рис. 1.11. Неструктурированное решение задачи определения максимума из трех чисел

Вначале сравниваются *a* и *b*. Если $a > b$, то претендентами на максимум остаются *a* и *c*, и максимум окончательно определяется из сравнения $a > c$. Если *a* не больше *b*, то максимум находится из сравнения *b* и *c*.

Как видно из рисунка 1.11, схема нашего решения не является композицией типовых структур. Раз так, запишем ее решение на Паскале с помощью `goto`. И уж если мы пока не можем обойтись в этой ситуации без переходов, пусть это будет примером того, как не надо программировать (листинг 1.9). Я запутал текст несколько больше обычного, хотя это не совсем беспорядочное программирование. Некоторый принцип все же применялся. Вначале я записал по порядку три действия, которые присваивают значение величине *M*, ведь именно одно из них мы все равно должны будем выполнить. Затем, двигаясь от трех этих действий, как от фундамента (снизу вверх по схеме), записал условия. Потом предусмотрел недостающие переходы.

Листинг 1.9. Так делать не надо!

```
goto 7;
2: M := a; goto 1;
3: M := c; goto 1;
4: M := b; goto 1;
5: if a>c then goto 2 else goto 3;
6: if c>b then goto 3 else goto 4;
7: if a>b then goto 5 else goto 6;
1:
```

Получилась еще одна порция спагетти.

Теорема структурирования

В статье, опубликованной в 1966 году, итальянские математики Коррадо Бём (Corrado Böhm) и Джузеппе Якопини (Guiseppe Jacopini) доказали, что любой алгоритм можно представить с помощью только двух базовых структур. Этот результат и носит название теоремы структурирования. Доказательство Бёма и Якопини конструктивно, то есть не только устанавливает саму возможность сведения алгоритма к двум базовым структурам, но и дает способ такого преобразования. Мы используем несколько иной вариант доказательства — преобразование Ашкрофта – Манны (E. Aschkroft, Z. Manna).

Теорема. *Для любой программы, выраженной блок-схемой с одним входом и одним выходом, можно построить эквивалентную программу, то есть выполняющую те же преобразования исходных данных в результаты с помощью тех же вычислений, единственными управляющими структурами в которой являются цепочка и цикл с предусловием.*

Покажем вначале, что структуру «ветвление» можно представить с помощью цепочки и цикла с предусловием. Запишем ветвление в общем виде:

```
if B then S1 else S2
```

Здесь `B` — логическое выражение (условие); `S1` и `S2` — операторы. Построим программу, эквивалентную этой конструкции, но использующую только цепочку (следование операторов друг за другом) и цикл **`while`**. Используем вспомогательные логические переменные `b1` и `b2` (будем считать, что таких переменных нет в исходной программе).

```
b1 := B; b2 := true;
while b1 do begin
   S1; b1 := false; b2 := false;
end;
while b2 do begin
   S2; b2 := false;
end;
```

Нетрудно убедиться, что эта программа выполняет те же действия и проверки и в том же порядке, что и исходная, оперируя также со вспомогательными переменными. Действительно, условие `B` вычисляется один раз, как и в исходной программе, до выполнения `S1` либо `S2`. Если `B` истинно, то выполняется первый цикл **`while`** один раз (поскольку `b1` внутри цикла получает значение **`false`**), а в нем оператор `S1`. При этом `b2` становится равным **`false`** и поэтому второй цикл не выполняется. Если `B` — ложь, то `b1` — тоже ложь, первый цикл не выполняется, но, поскольку `b2` в этом случае сохраняет значение **`true`**, выполняется ровно один раз второй цикл, а в нем `S2`.

Понятно, что такая замена бессмысленна на практике, она имеет только теоретическое значение.

Продолжим доказательство, формулируя алгоритм преобразования исходной блок-схемы в эквивалентную структурированную программу. Для иллюстрации алгоритма будем использовать схему из рассмотренного примера о нахождении максимума из трех чисел (см. рис. 1.11).

1. Пронумеруем блоки исходной схемы (рис. 1.12). Блоку, с которого начинается выполнение, дадим номер 1. Выход из схемы обозначим нулем. Остальные блоки пронумеруем произвольно. На рисунке 1.12 внутри каждого блока записано обозначение его действия (`S`) или проверяемого условия (`B`), сопровождаемое номером блока. Эти обозначения использованы вместо исходных действий и условий (см. рис. 1.11), чтобы не привязывать рассмотрение к конкретной задаче и облегчить ссылки на действия и проверки каждого блока по их номерам.
2. Введем в рассмотрение вспомогательную переменную целого типа — программный счетчик. Обозначим ее `L`, считая, что в исходной программе такой переменной нет.

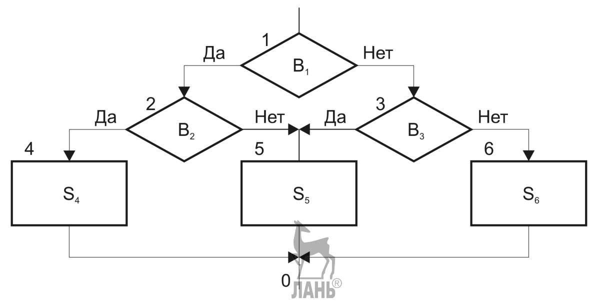


Рис. 1.12. Структурирование алгоритма нахождения максимума из трех чисел

3. Для каждого функционального блока S_i запишем заменяющий оператор вида

```
begin Si; L := <номер блока, следующего за Si> end
```

В нашем примере это будут операторы
для блока 4:

```
begin S4; L := 0 end
```

для блока 5:

```
begin S5; L := 0 end
```

для блока 6:

```
begin S6; L := 0 end
```

4. Для каждого блока проверки B_i запишем заменяющий оператор вида

```
if Bi then L := <номер следующего блока в ветви "Да">
else L := <номер следующего блока в ветви "Нет">
```

Для блока 1 в примере:

```
if B1 then L := 2 else L := 3
```

Для блока 2:

```
if B2 then L := 4 else L := 5
```

Для блока 3:

```
if B3 then L := 5 else L := 6
```

5. Построим программу в виде цепочки из присваивания `L := 1` и цикла **while** с условием `L<>0`. Внутри цикла поместим много-

значное ветвление (можно вложенные **if** – **then** – **else**) по значению L с числом ветвей, равным числу блоков. В ветви, соответствующей данному номеру блока, помещается заменяющий оператор этого блока. Для нашего примера получится такая программа:

```
L := 1;
while L<>0 do
   case L of
   1: if B1 then L := 2 else L := 3;
   2: if B2 then L := 4 else L := 5;
   3: if B3 then L := 5 else L := 6;
   4: begin S4; L := 0 end;
   5: begin S5; L := 0 end;
   6: begin S6; L := 0 end;
   end
```

Эта программа выполняет те же действия и те же проверки в том же порядке, что и исходная, то есть эквивалентна ей. Программа построена с помощью следования, цикла с предусловием, развилки и многозначного ветвления. Многозначное ветвление может быть заменено вложенными **if** – **then** – **else**, а эти конструкции, как было показано выше, могут быть заменены цепочками и циклами **while**. После таких замен получится программа, состоящая только из цепочек и циклов с предусловием. Что и требовалось доказать.

Полученная эквивалентная программа, особенно если сделать подстановки вместо **case** и **if** – **then** – **else**, получается очень громоздкой и, хотя является структурированной, но, пожалуй, даже менее понятна, чем программа из листинга 1.9.

Чтобы получить практически полезный результат, то есть решить исходную задачу нахождения максимума трех чисел с помощью структурированной и несложной программы, продолжим преобразования для нашего примера.

Заменим в программе присваивания L := 4, L := 5 и L := 6 на действия, выполняющиеся на следующем витке цикла при соответствующем значении L (помеченные этим значением L). Поскольку L больше не может принять значения 4, 5 и 6, то соответствующие ветви в **case** больше не нужны. Получим:

```
L := 1;
while L<>0 do
   case L of
```

```
1: if B1 then L := 2 else L := 3;
2: if B2 then begin S4; L := 0 end
   else begin S5; L := 0 end;
3: if B3 then begin S5; L := 0 end
   else begin S6; L := 0 end;
end
```

Аналогичную замену выполним теперь для `L := 2` и `L := 3`. Получим:

```
L := 1;
while L<>0 do
   case L of
   1: if B1 then
           if B2 then begin S4; L := 0 end
           else begin S5; L := 0 end
      else
           if B3 then begin S5; L := 0 end
           else begin S6; L := 0 end;
   end
```

Теперь остался только один вариант в операторе **`case`**, а значит, сам этот оператор больше не нужен, поскольку никакого значения, отличного от 1, счетчик при входе в цикл принять не может. После выполнения этого единственного варианта счетчик всегда принимает значение 0, а значит, цикл выполняется ровно один раз, и тоже больше не нужен. Не требуется и сам счетчик. Окончательно, имеем:

```
if B1 then
   if B2 then S4
   else S5
else
   if B3 then S5
   else S6
```

После подстановки вместо `S` и `B` соответствующих действий и условий исходной задачи получаем ее структурированное и вполне приемлемое решение:

```
if a>b then
   if a>c then M := a
   else M := c
else
   if c>b then M := c
   else M := b
```

Конечно, так можно было записать и не прибегая к громоздким формальным преобразованиям. Достаточно было заметить, что, дублируя блок 5 (`M := c`), мы превращаем схему в композицию трех развилок.

Продемонстрированное преобразование не может, конечно же, рассматриваться как прием для практической работы: создали алгоритм «как попало», а потом преобразовали в структурированную форму. Нужно стремиться с самого начала при разработке программы мыслить структурно. При использовании современных языков это несложно. Доказанная же теорема позволяет нам быть уверенными, что структурное решение всегда существует.

Хочу также подчеркнуть, что структурный подход отнюдь не гарантирует получения наилучших решений поставленных задач. Ничто не заменит изобретательности, интуиции, фундаментальной подготовки. Структурное программирование лишь позволяет выразить уже найденный алгоритм в ясной форме. Вернемся к нашей задаче о поиске максимума трех чисел. Ее решение может быть и другим. Снимая ограничение на допустимость только одиночных сравнений, можем записать:

```
if (a>b) and (a>c) then
   M := a
else if b>c then
   M := b
else
   M := c
```

Стилистически это решение, безусловно, лучше предыдущего: короче и понятней. А если действует короткая схема вычисления логических выражений, то и число сравнений всегда равно двум, как в предыдущем случае. А вот еще вариант:

```
M := a;
if b>M then M := b;
if c>M then M := c;
```

Здесь использован совсем другой подход к задаче. И никакое структурное программирование найти такой подход не помогает. Хотя в этом варианте может потребоваться больше одного присваивания, зато подобный способ решения, в отличие от предыдущих, легко обобщается на случай произвольного числа переменных.

Пошаговая детализация

Мы рассмотрели структурное программирование в его собственном смысле — программирование на основе стандартных управляющих структур. Но этот термин часто употребляется более широко, обозначая весь комплекс технологических приемов, сформировавшихся в конце 1960-х — начале 1970-х годов. К числу таких приемов относится и метод пошаговой детализации при разработке программ[10].

Суть этого метода состоит в том, что, приступая к решению задачи, ее вначале рассматривают как некое единое действие. Затем выделяют в задаче небольшое число подзадач и устанавливают характер их взаимодействия. Далее каждая подзадача разбивается снова на несколько подзадач, и так до тех пор, пока выделенные подзадачи не смогут быть решены непосредственно. При разбиении на подзадачи обычно руководствуются правилом «7±2», разделяя каждую задачу примерно на 5–9 частей. Считается, что именно таким количеством элементов человек может свободно оперировать, не утрачивая контроля над их взаимодействием.

Уже после первых шагов детализации может быть записана программа и начата ее отладка. Нереализованные части программы временно заменяются «заглушками», которые обеспечивают некоторый минимум действий вместо той части, которую замещают.

Структурное программирование служит теоретической базой метода пошаговой детализации, гарантируя, что процесс детализации всегда возможен.

На первом шаге детализации задача предстает как один функциональный блок (рис. 1.13). Далее этот блок может быть представлен как одна из базовых структур, содержащая в свою очередь несколько блоков. Каждый функциональный блок этой структуры вновь может быть представлен как одна из типовых структур. И так далее, до тех пор, пока все получившиеся функциональные блоки не будут реализовываться с помощью отдельных операторов выбранного языка программирования.

---

[10] Разделом структурного программирования есть основания считать и объектно-ориентированное программирование (ООП). Основополагающая работа О.-Й. Даля (O.-J. Dahl), создателя первого объектно-ориентированного языка Симула-67, и Ч. Э. Р. Хоара «Иерархические структуры программ», в которой обсуждаются основные понятия ООП, такие как классы объектов, была опубликована в 1972 году в книге под названием «Структурное программирование».

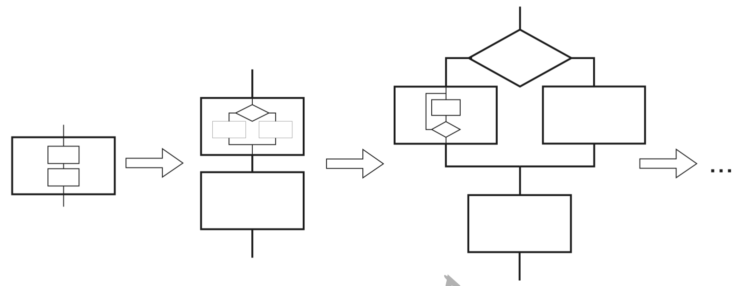


Рис. 1.13. Пошаговая детализация

Можно даже предложить универсальный алгоритм решения любой задачи (больше в шутку, но и всерьез):

```
Нарисуй прямоугольник и обозначь выполняемое им
действие;
while есть прямоугольник, действие которого
   не является простым
do begin
   Выбери на рис. 1.9 подходящую структуру,
   подставь ее вместо прямоугольника и
   обозначь условия и действия ее блоков
end;
{ Задача решена }
```

Буквальное применение такого приема — замена на каждом шаге детализации функционального блока ровно на одну структуру — непрактично. Получается слишком много слишком мелких шагов. Это лишь предельный случай, непосредственно следующий из идеи структурного программирования.

Совсем не обязательно пользоваться при разработке программ с помощью пошаговой детализации блок-схемами. Наоборот, унификация стиля и достаточная выразительность структурных управляющих операторов, небольшое число подзадач, выделяемых на каждом шаге, делают блок-схемы ненужными. Они уже не дают большей наглядности, чем обеспечивает хорошо структурированная программа.

Пошаговая детализация имеет и другие названия. В статье «Разработка программ пошаговым усовершенствованием», опубликованной в 1971 году, Н. Вирт, вероятно, впервые вводит подобный термин. Называют такой подход также нисходящим проектированием, разработкой по методу «сверху вниз» и т. д.

Все гениальное просто

«Будь попроще, дурачок!» — такой призыв стал одним из лозунгов в борьбе за улучшения стиля программ. По-английски это звучит так: «Keep It Simple Stupid», сокращенно KISS[11]. Провозглашен был KISS-принцип, который вошел в программистский жаргон одновременно со структурным программированием. В те годы появилось много книг, посвященных технологии программирования. Кроме собственно структурного программирования и пошаговой детализации в этих книгах давалось множество рекомендаций по улучшению стиля. Речь шла и о размещении текста программы, использовании комментариев и выборе имен переменных, оптимальном размере модуля, отказе от изощренных приемов и много еще о чем. В конечном итоге все эти рекомендации направлены на *упрощение* чтения и понимания программы путем стандартизации и *упрощения* ее структуры и стиля. Так что принцип этот вполне фундаментальный, хоть и несерьезный по форме. Он и теперь не потерял своей актуальности — впрочем, у нас еще будет повод вернуться к разговору о простом и сложном.

Успехи структурного программирования

После определенного периода дискуссий на тему о допустимости `goto` идеи структурного программирования были приняты мировым программистским сообществом. Получили широкую известность результаты первых проектов, выполненных по технологии структурного программирования. Внимание привлек опыт разработки информационной системы для газеты «Нью-Йорк Таймс». В проекте, которым руководил Харлан Миллз (Harlan Mills), был также использован новый способ организации программистского коллектива, так называемая «бригада главного программиста». Основную работу по написанию программы в такой бригаде выполняет бригадир. Остальные обеспечивают ему необходимые условия и решают вспомогательные задачи. Опыт этой разработки подтвердил преимущества структурного программирования. Трудоемкость создания программы размером более 83 000 строк составила 11 человеко-лет, а первая ошибка проявила себя только через 13 месяцев после начала эксплуатации системы.

[11] Kiss (англ.) — поцелуй, целоваться.

Языки программирования, появившиеся после 1968 года, обязательно содержали достаточные средства для структурного программирования, хотя во многих из них оператор перехода все еще был предусмотрен. Такие языки, как Паскаль и Си, к обсуждению которых мы переходим, иногда даже называют языками структурного программирования, хотя цели создателей этих языков, конечно же, не сводились к правильному оформлению ветвлений и циклов.

Паскаль

Создатель языка Паскаль — Никлаус Вирт (Niklaus Wirth). После отклонения рабочей группой ИФИП его проекта усовершенствования Алгола-60, Н. Вирт, работая в Стэнфордском университете (США), осуществил реализацию предложенного им языка на компьютере IBM/360. Этот язык в дальнейшем получил название Алгол W и применялся в ряде университетов для целей обучения.

В 1967 году Н. Вирт вернулся в родную Швейцарию, а через год организовал группу из трех помощников (У. Амман (U. Ammann), Э. Мармье (E. Marmier), Р. Шилд (R. Schild)) для реализации языка, который мы знаем теперь под именем Паскаль. Разработка Паскаля преследовала несколько целей. Во-первых, это стремление получить язык, пригодный для систематического обучения программированию и способствующий формированию правильного стиля. Во-вторых, язык должен был стать инструментом для разработки системных программ (компиляторов, операционных систем). В-третьих, при проектировании было обращено внимание на возможность компактной и эффективной реализации.

В Паскале нашли дальнейшее развитие идеи Алгола-60 и Алгола W. Одной из важнейших новаций языка была последовательно, ясно и полно реализованная концепция типов данных. Она стала практическим воплощением теории структурной организации данных, активно развивавшейся в те годы Ч. Э. Р. Хоаром.

Предварительное описание Паскаля было опубликовано в 1968 году. Первый компилятор (написанный на самом Паскале) заработал в 1970 году. С 1972 года Паскаль стал использоваться в ETH при обучении программированию. В 1973 году в язык были внесены некоторые изменения и было опубликовано Пересмотренное сообщение о языке. Этот авторский вариант языка мы и будем называть Паскалем или стандартным Паскалем, в отличие от многочисленных последующих модификаций.

Первый компилятор не обеспечивал генерации машинного кода, сопоставимого по эффективности с кодом, получаемым для такой же машины трансляцией с Фортрана. Поэтому была предпринята еще одна реализация, осуществленная Урсом Амманом и обеспечившая получение эффективных программ. Этот компилятор был распространен во многих университетах и в промышленности.

Однако по-настоящему широкую известность Паскалю принесла реализация, выполненная в ETH и основанная на трансляции программы в П-код (в оригинале — P-code) — машинный код не существующей реально, а спроектированной специально для этих целей вычислительной машины. Работа такой машины, исполняющей П-код, имитируется программой-интерпретатором. Такой подход позволяет получить транслятор, легко переносимый на разные компьютеры и операционные системы. Для переноса достаточно написать сравнительно несложную программу-интерпретатор. Именно с этим проектом связано само понятие П-кода (П — от Паскаль), который в дальнейшем неоднократно использовался при реализации языков программирования.

Цели, поставленные Н. Виртом при создании Паскаля, без сомнения, были достигнуты. В Паскале впервые была последовательно реализована концепция строгой статической типизации, предусматривающая возможно более полный контроль соответствия типов на стадии компиляции. Последующее развитие показало правильность этой концепции. Некоторые языки в своей эволюции прошли путь от почти полного ее отрицания до признания и развития. Идеи, впервые воплощенные в Паскале, нашли продолжение в последующих, «паскалеподобных» языках: Ада, Модула-2, Оберон. И даже такие, казалось бы, не похожие внешне на Паскаль языки, как Ява и Си#, впитали в себя его концепции.

В 1982 году был принят стандарт ISO на язык Паскаль, а в 1990 году — его новая редакция. Однако этот стандарт не оказал существенного влияния на развитие языка.

В нашей стране Паскаль заметили с опозданием. Во второй половине 70-х в центре внимания были ЕС ЭВМ (клоны IBM/360) с их Фортраном и ПЛ/1. Многие ведущие специалисты по программированию, вышедшие из математической среды, явно отдавали предпочтение Алголу-68. Написанное в 1974 году классическое руководство по Паскалю Н. Вирта и К. Йенсен было издано по-русски лишь в 1982

году [Йенсен, 1982], став первой общедоступной книгой по языку. Хотя к этому времени уже существовали реализации Паскаля на БЭСМ-6, Эльбрус-1 и ЕС ЭВМ, они оставались неизвестными для многих программистов. Широкую известность Паскаль приобрел лишь в первой половине 1980-х годов с появлением микрокомпьютеров и выходом книг по языку.

Основные черты языка Паскаль

Ниже приводятся основные характеристики авторской версии языка Паскаль. В позднейших диалектах, в частности, в системах программирования компании Borland, в язык вносились многочисленные изменения.

- Программа записывается единым блоком. Раздельная трансляция частей программы не предусмотрена. Модули отсутствуют.
- Обязательность описаний. Строгий контроль соответствия типов.
- Стандартные типы данных: целый (integer), вещественный (real), символьный (char), логический (Boolean). Перечислимые и ограниченные типы.
- Способы организации данных: массивы произвольной размерности (линейные, двумерные и т. д.), но только фиксированного размера; записи, в том числе с вариантами; множества; последовательные файлы.
- Указатели и основанное на их использовании динамическое распределение памяти.
- Управляющие операторы: **`if`** – **`then`** – **`else`**, **`case`** – **`of`** – **`end`**, **`for`** – **`to`** (**`downto`**) – **`do`**, **`while`** – **`do`**, **`repeat`** – **`until`**, **`goto`**.
- Процедуры и функции с возможностью вложения и рекурсии. Параметры передаются по значению или по ссылке (параметры-переменные).

Сортировка вставками

Рассмотрим несколько вариантов алгоритма сортировки простыми вставками, который используется в качестве примера при обсуждении других языков.

Пусть требуется расположить в порядке неубывания[12] `n` элементов массива `a`, содержащего вещественные числа.

[12] Речь идет о сортировке «по неубыванию», а не о сортировке «по возрастанию», поскольку в массиве могут быть элементы с одинаковыми значениями.

Запишем описание типа массива[13]:

```
const
   nmax = 10000;
type
   tArray = array [1..nmax] of real;
```

Сортировка вставками — это простой и очень естественный алгоритм упорядочения. Именно таким способом мы часто выполняем сортировку вручную.

Для упорядочения по неубыванию элементов массива поступим так (рис. 1.14): выберем второй по порядку элемент и разместим его должным образом по отношению к первому. То есть если значение второго элемента больше или равно значению первого, то второй оставляем на своем месте, если нет — расположим его перед первым, сдвигая первый на второе место. Теперь первые два элемента упорядочены. Берем следующий, третий, и *вставим* его в нужное место по отношению к первым двум. Теперь упорядочены первые три. Берем поочередно следующие, вплоть до последнего и помещаем каждый в нужное место уже упорядоченной последовательности предыдущих. Отыскивая подходящее место для данного элемента, можно просто последовательно просматривать предшествующие, начиная с того, который расположен перед выбранным. Как только встречается элемент со значением меньшим или равным выбранному, помещаем выбранный сразу за таким элементом. В процессе просмотра и движения в сторону меньших номеров (влево на схеме) сдвигаем вправо на одну позицию каждый элемент, который будет располагаться правее выбранного в упорядоченном массиве.

Запишем этот алгоритм на Паскале. Обозначим i — номер выбранного элемента; x — значение i-го элемента.

```
for i := 2 to n do begin
   x := a[i];
   { Вставить x в последовательность
   предшествующих ему элементов }
end;
```

[13] При выборе названия типа для массива я использовал принцип «венгерской нотации». Первая строчная буква *t* в имени типа `tArray` подчеркивает природу этого наименования (*t* — от **`type`**). Такой принцип обозначений был предложен сотрудником Microsoft, венгром по происхождению, Чарльзом Симонаи (Charles Simonyi).

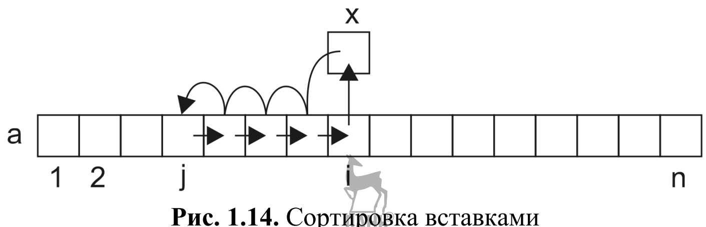


Рис. 1.14. Сортировка вставками

Детализируя вставку *i*-го элемента в последовательность элементов с номерами от *i*-1 до 1, обозначим *j* — номер элемента, с которым сравнивается значение *x* (*i*-й элемент). Оформляя решение как процедуру, получаем:

```
{ Сортировка простыми вставками }
procedure InsSort(var a: tArray; n: integer);
var
   i, j : integer;
   x    : real;
begin
   for i := 2 to n do begin
      x := a[i];          { выбор элемента }
      j := i - 1;
      while (j>0) and (x<a[j]) do begin
         a[j+1] := a[j]; { сдвиг вправо   }
         j := j - 1;     { движение влево }
      end;
      a[j+1] := x;       { x попадает на свое место }
   end;
end;
```

Использованная запись условия в цикле **`while`** предполагает, что действует короткая схема вычисления логических выражений, то есть, если условие `(j>0)`, записанное перед **`and`**, не выполняется, то условие, записанное после **`and`**, вообще не проверяется, а значит, не происходит и обращения к несуществующему (нулевому) элементу массива. Левая часть составного условия как бы защищает правую от вычисления при недопустимом значении `j`.

Можно попытаться ускорить работу программы, а заодно избавиться от опасности обращения к несуществующему элементу массива. Для этого дополним массив нулевым элементом, для чего придется изменить описание типа:

```
type
   tArray = array [0..nmax] of real;
```

Сортировать при этом по-прежнему будем элементы с номерами от 1 до n, а элемент `a[0]` будет исполнять роль барьера. Перед каждым циклом вставки будем помещать в элемент `a[0]` значение `x`. Это позволит упростить условие внутреннего цикла, отказавшись от проверки `(j>0)`. Корректное завершение цикла при этом гарантируется тем, что при сравнении `x` с `a[0]` условие `x<a[j]` нарушается, поскольку при `j=0` значение `x` будет равно `a[0]`. Барьер не преодолеть. Проверка условия цикла выполняется многократно, и его упрощение позволяет надеяться на ускорение работы программы.

```
{ Сортировка вставками с барьером }
procedure InsSort(var a: tArray; n: integer);
var
   i, j : integer;
   x    : real;
begin
   for i := 2 to n do begin
      x := a[i];
      a[0] := x;  { установка барьера }
      j := i - 1;
      while x<a[j] do begin
         a[j+1] := a[j]; { сдвиг вправо   }
         j := j - 1;     { движение влево }
      end;
      a[j+1] := x;       { x попадает на свое место }
   end;
end;
```

Наконец, можно заметить, что от переменной `x` можно вообще отказаться, сэкономив на присваивании ей значения. Роль `x` будет исполнять `a[0]`. При этом можно рассчитывать, что время обращения к элементу массива `a[0]` будет не больше времени доступа к неиндексированной переменной `x`. Большинство компиляторов обращается с элементами массивов, имеющими константные индексы, как с простыми переменными.

```
{ Сортировка вставками с барьером. Вариант 2 }
procedure InsSort(var a: tArray; n: integer);
var
   i, j : integer;
begin
```

```
    for i := 2 to n do begin
       a[0] := a[i];        { установка барьера }
       j := i - 1;
       while a[0]<a[j] do begin
          a[j+1] := a[j]; { сдвиг вправо   }
          j := j - 1;     { движение влево }
       end;
       a[j+1] := a[0]; { a[i] попадает на свое место }
    end;
 end;
```

При упорядочении массивов с большим числом элементов сортировка вставками неэффективна. Количество выполняемых ею действий в среднем пропорционально n^2, что является плохим показателем для алгоритмов сортировки. Но если число элементов не превышает 20, да к тому же эти элементы частично упорядочены, то сортировка вставками работает быстрее других алгоритмов. Поэтому ее используют при построении усовершенствованных алгоритмов в качестве вспомогательного средства для сортировки коротких частей массива. Для такого случая можно предусмотреть еще одну оптимизацию.

Если массив частично отсортирован, то нередко будет встречаться ситуация, когда выбранный i-й элемент массива возвращается на свое же место, поскольку оказывается больше или равен предыдущему элементу. При этом происходят несколько ненужных в этом случае присваиваний. Предусмотрев отдельную проверку для сравнения i-го элемента с предыдущим, лишних присваиваний можно избежать.

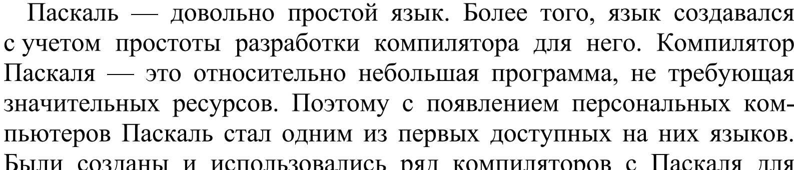
```
 {Сортировка короткого частично упорядоченного массива}
 procedure InsSort(var a: tArray; n: integer);
 var
    i, j : integer;
    x    : real;
 begin
    for i := 2 to n do begin
       x := a[i];
       if x<a[i-1] then begin
          a[i] := a[i-1];
          j := i - 2;
          while (j>0) and (x<a[j]) do begin
             a[j+1] := a[j]; { сдвиг вправо   }
             j := j - 1;     { движение влево }
          end;
```

```
            a[j+1] := x;        {x попадает на свое место}
          end;
      end;
  end;
```

Хотя выигрыш от этой модификации не будет большим, получившаяся программа хороша для нас тем, что кроме циклов **`for`** и **`while`**, включает и конструкцию **`if`**, и, при использовании в качестве примера, полнее иллюстрирует средства обсуждаемых языков программирования.

К этому варианту можно применить идею барьера, но, если имеется в виду сортировка подмассивов, то для барьера нет места.

Turbo Pascal



Паскаль — довольно простой язык. Более того, язык создавался с учетом простоты разработки компилятора для него. Компилятор Паскаля — это относительно небольшая программа, не требующая значительных ресурсов. Поэтому с появлением персональных компьютеров Паскаль стал одним из первых доступных на них языков. Были созданы и использовались ряд компиляторов с Паскаля для микрокомпьютеров.

Однако настоящую революцию произвело появление в 1983 году Turbo Pascal — системы, которая по своей организации была новой[14] не только для компиляторов Паскаля, но и для систем программирования вообще. Основой Turbo Pascal был компактный и очень быстрый компилятор, высокая скорость работы которого обеспечивалась за счет выполнения основных действий в оперативной памяти и однопроходной схемы трансляции, при которой компиляция выполняется за один просмотр транслятором исходного текста программы. В отличие от традиционных компиляторов, которые всегда формировали файл объектного кода, Turbo Pascal обеспечивал такой режим трансляции, когда машинный код формировался прямо в памяти и мог быть немедленно исполнен. Система содержала встроенный текстовый редактор, взаимодействующий с компилятором. Подготовленная с помощью редактора программа нажатием пары клавиш могла быть откомпилирована и, если ошибок не обнаруживалось, тут же запущена. При обнаружении ошибки происходил возврат в режим

[14] Существует мнение, что Turbo Pascal унаследовал многие черты системы UCSD Pascal.

редактирования, и курсор устанавливался на место ошибки. Текст программы также по возможности хранился в памяти, что минимизировало обращения к дискам и также ускоряло работу. Интегрированный редактор и высокая скорость компиляции обеспечивали не меньшую оперативность, чем при использовании интерпретирующих систем. В то же время компиляция в машинный код, в отличие от интерпретации, обеспечивала получение быстрых машинных программ.

Turbo Pascal был создан молодым датчанином Андерсом Хейльсбергом (Anders Hejlsberg). Под названием Kompass Pascal программа продавалась в Европе, но, видимо, без особого успеха. Права на программу были приобретены предприимчивым французом Филиппом Каном (Philippe Kahn), математиком по образованию, учившимся некоторое время в Цюрихе у Н. Вирта. Он переехал в США, где создал компанию Borland и организовал продажи разработанного Хейльсбергом компилятора под названием Turbo Pascal. Благодаря удачной рекламе, невысокой цене ($49.95) и выдающимся техническим характеристикам программа быстро приобрела популярность. Коммерческий успех первой версии Turbo Pascal позволил компании Borland активно заниматься развитием системы. Основным разработчиком новых версий по-прежнему был А. Хейльсберг.

Игнорируя стандарт языка Паскаль, Borland в каждой новой версии Turbo Pascal и в последовавших за ним системах Borland Pascal и Delphi вносила существенные изменения в язык. Чтобы сохранять совместимость с предыдущими версиями, эти изменения, за небольшим исключением, сводились к расширению языка, который со временем стал большим и сложным. С выходом системы Delphi язык стал называться Объектный Паскаль[15] (Object Pascal).

Компания Borland не сочла целесообразным и переходить в своих системах от языка Паскаль к языку Модула-2, который был создан Н. Виртом как наследник Паскаля — он не обеспечивал обратной совместимости с Паскалем[16]. Несмотря на то, что система програм-

[15] Язык с названием Object Pascal впервые был разработан и реализован в компании Apple Computer с участием Н. Вирта в середине 80-х годов. Объектный Паскаль компании Borland — это другой язык.

[16] Новый язык тем и отличается от модернизированного старого, что в нем не только имеются нововведения, но и отсутствуют многие конструкции языка-предшественника. За счет утраты обратной совместимости это позволяет не усложнять язык.

мирования на языке Модула-2 была уже разработана компанией, Borland решила не выпускать ее на рынок, а продолжила дальнейшее расширение своих версий Паскаля. Turbo Pascal for Windows был первым компилятором, работающим целиком под управлением Windows. Однако он не облегчал радикально создание программ для этой операционной системы. Программировать пользовательский интерфейс приходилось вручную, отсутствовал встроенный отладчик.

Другое дело — появление в 1995 году системы Delphi. Это была первая система программирования, сочетавшая компилятор в настоящий машинный код и средства визуального программирования, которые принципиально упрощали создание программ для Windows. Хотя аналогичные средства быстрой разработки программ существовали в системе Visual Basic уже с 1991 года, Delphi стала весьма популярной благодаря возможности получения эффективных программ и более приемлемому для многих программистов языку.

Что касается языка Объектный Паскаль, то он, подобно Visual Basic, не может, по существу, рассматриваться как универсальный машинно-независимый язык. Он не соответствует каким-либо международным стандартам[17], и его развитие в значительной мере определяется не универсальными потребностями, а необходимостью приспособления к частным технологиям.

Позже компанией Borland была выпущена совместимая с Delphi система Kylix, предназначенная для разработки программ на Объектном Паскале для операционной системы Linux. Однако распространения она не получила.

Язык Си

Язык Си (C — «си», третья буква алфавита английского языка) создан в ходе работ по операционной системе UNIX, разработка которой велась в Bell Laboratories. Язык разработан и реализован на миникомпьютере PDP-11 в 1972 году Деннисом Ритчи (Dennis Ritchie).

Название языка происходит от его предшественников языков BCPL и B («би» — вторая буква в алфавите). Язык B был разработан и ис-

[17] Существует мнение, что именно отсутствие необходимости придерживаться стандартов способствовало тому, что система визуального программирования была создана компанией Borland на базе языка Паскаль, а не Си++. По моему мнению, это обусловлено еще и тем, что руководил разработкой основанных на Паскале систем (вплоть до Delphi 2.0) выдающийся программист А. Хейльсберг.

пользован Кеном Томпсоном (Ken Thompson) при создании первого варианта UNIX. BCPL и B — бестиповые языки, которые предназначались в первую очередь для манипуляций с находящимися в памяти данными безотносительно к их природе. Такая потребность неизбежно возникает при разработке системных программ. Си и создавался в первую очередь как инструмент для разработки операционной системы UNIX. И хотя типы в языке Си имеются, он унаследовал от BCPL и B весьма либеральное к ним отношение, предоставляя программисту максимальную свободу для манипуляций данными, не ограниченную жесткими правилами совместимости.

Хотя Д. Ритчи не называет другие языки, кроме BCPL и B, в качестве предшественников, в языке Си можно увидеть черты Алгола-60 и Алгола-68.

Создавая инструмент для собственных разработок, Д. Ритчи и К. Томпсон ставили на первое место гибкость этого инструмента. Язык должен был давать возможность выполнять любые манипуляции с данными, которые доступны при программировании на языке ассемблера. При этом, в отличие от языка ассемблера, должны быть обеспечены большая выразительность и удобство при записи программы. В частности, необходимыми элементами являются достаточные средства структурного программирования и удобные правила оформления подпрограмм и передачи параметров. Именно таким и оказался язык Си. Однако гибкость при обращении с данными достигнута за счет отказа от высокоуровневых механизмов контроля, в частности, контроля соответствия типов. Ответственность за это соответствие возлагается за программиста. Си, без сомнения, предоставляет больше возможностей делать ошибки, чем языки, основанные на строгой типизации.

Отсутствие высокоуровневых механизмов контроля и высокоуровневых абстракций, использование в языке понятий, таких как «адрес», характерных для машинного языка, позволяет считать Си языком среднего уровня, занимающим промежуточное положение между языками низкого (языками ассемблера) и языками высокого уровня. Н. Вирт характеризует Си как синтаксически усовершенствованный ассемблер. Это совсем не обидная для языка оценка. Язык с самого начала предназначался для такой роли и для тех применений (разработка операционных систем), где традиционно в начале 1970-х годов

использовался ассемблер. Важным преимуществом Си перед языками ассемблера является его независимость от конкретной машины.

Язык Си достаточно прост. Это позволяет создать компилятор для него без чрезмерных затрат времени и средств. Конструкции языка таковы, что их реализация оказывается весьма эффективной. В целом язык Си представляется весьма гармоничной конструкцией, ориентированной на определенный круг задач.

В 1973 году система UNIX была переписана на Си. Лишь несколько сот строк ядра операционной системы оставались запрограммированны на ассемблере. С этого времени Си и UNIX неразрывно связаны. Разработчики UNIX и Си предприняли специальные усилия по созданию переносимой версии языка и компилятора для него. Благодаря этому стала мобильной и была перенесена на компьютеры различных архитектур и ОС UNIX.

В конце 70-х, благодаря распространению UNIX, язык Си становится широко известным и популярным. Этому способствовал и выход великолепной книги по языку, написанной Д. Ритчи в соавторстве с Брайаном Керниганом (Brian Kernighan) [Керниган, 1985]. Книга Кернигана и Ритчи стала классикой жанра и даже получила специальное сокращенное название K&R. Описание языка, данное в этой книге, в течение последующих лет исполняло роль фактического стандарта. Хотя описание Си в K&R оставалось не вполне строгим и формальным.

В 1989 году Американским Национальным Институтом Стандартов (ANSI) был принят стандарт Си, получивший повсеместное признание как эталон языка. В 1990 этому стандарту был придан статус международного (ISO/IEC 9899:1990). ANSI Си является одним из наиболее переносимых языков программирования, реализованным практически на любых компьютерах. Он часто используется как промежуточный язык при реализации других языков. Разработка для нового языка конвертора в ANSI Си вместо компилятора в машинный код облегчает задачу создателей транслятора и одновременно обеспечивает переносимость нового языка на многие платформы. При этом можно заметить, что недостаточная надежность Си не играет в этом случае никакой роли, поскольку программы на Си, сгенерированные автоматически, заведомо избавлены от языковых ошибок (при условии, что сам конвертор работает правильно).

В 1999 году Международной организацией по стандартизации (ISO) принят обновленный стандарт языка Си — ISO/IEC 9899:1999.

Основные черты языка Си

- Компактность записи. В языке предусмотрены краткие способы записи многих часто употребляемых действий. Ярким примером является изящная возможность записывать `i++;` вместо `i = i + 1;` Запись `x += 2;` означает подобно Алголу-68 `x = x + 2;` (знак «=» обозначает присваивание).
- Язык, подобно Алголу-68, стимулирует побочные эффекты. Операторы могут использоваться как выражения, а выражения — как операторы. Например, запись `i++;` (с точкой с запятой) — это оператор, но такая же конструкция может записываться в выражении. Например, `j + i++` означает добавление к значению `j` значения переменной `i`, которое эта переменная имела до вычисления выражения, и последующее увеличение значения `i` на единицу. Такие возможности позволяют получить очень компактную запись, но существенно усложняют понимание программы.
- Типы данных: целые (`int`) с вариантами коротких (`short`), длинных (`long`) и беззнаковых (`unsigned`), вещественные одинарной (`float`) и двойной точности (`double`), символьный (`char`). Символьный и целый типы взаимозаменяемы, к символам применимы те же операции, что и к целым. Отсутствие логического типа (вместо него используется целый) провоцирует ошибки при записи условий. Важную роль играют указатели.
- Богатый набор операций, распределенных по приоритету выполнения на 15 групп. Среди них поразрядные логические операции, операции сдвига, операция получения адреса. Адресная арифметика — арифметические операции, применимые к указателям.
- Ослабленный контроль соответствия типов.
- Единственный вид подпрограмм (в авторском варианте Си, K&R) — функции. В ANSI Си предусмотрены функции, не возвращающие значения. Параметры передаются только по значению. Для параметра-массива передается (по значению) его адрес. Вложенность функций отсутствует. Разрешена рекурсия.
- Способы организации данных: массивы, структуры (записи), объединения. Массивы и указатели практически неразличимы.
- Достаточный набор структурных управляющих операторов: `if`, `switch`, `while`, `for`, `do`. Операторы перехода `goto`, разрыва `break`, продолжения `continue`, возврата `return`.

- Программа состоит из отдельных файлов, которые являются единицами независимой компиляции.
- Перед трансляцией программа обрабатывается препроцессором. Он выполняет макроподстановки в текст программы, включение файлов в исходную программу и условную трансляцию.
- В языке отсутствуют операции ввода-вывода и операции со строками. Эти средства обеспечиваются библиотечными функциями. Стандарт ANSI Си определяет набор стандартных библиотек.

Примеры программ на языке Си

Книга Б. Кернигана и Д. Ритчи начинается с рассмотрения программы «Hello, World». Обсудим такую программу и мы.

Программа состоит из единственной функции с именем `main`. Выбор имени функции не случаен. Выполнение программы на Си всегда начинается с функции `main`. В нашем примере это функция целого типа, поскольку явно тип не указан, а по умолчанию принимается тип `int`. Оператор возврата (`return`), формирующий значение функции, при этом отсутствует.

Компилятор может выдать предупреждение по этому поводу, но по правилам Си это не считается ошибкой. Функцию можно вызывать как процедуру, при этом результат, возвращаемый функцией, просто отбрасывается. Фигурные скобки играют такую же роль как слова **begin** и **end** в Алголе-60 и Паскале. Единственный оператор в программе — вызов функции `printf` из стандартной библиотеки ввода-вывода `stdio`, которая присоединена к программе по умолчанию. Буква `f` в названии функции означает форматный вывод. Обратите внимание на запись «\n» перед закрывающей кавычкой. Это обозначение символа перехода на новую строку.

Сортировка вставками на языке Си

Рассмотрим последний из обсуждавшихся в разделе о языке Паскаль вариант сортировки простыми вставками. На Си это выглядит так:

```
/* Сортировка простыми вставками */
InsSort(float a[], int n) {
   int i, j; float x;
   for( i = 1; i<n; i++ )
      if( (x=a[i])<a[i-1] ) {
         a[i] = a[i-1];
         for( j = i-2; j>=0 && x<a[j]; j-- )
```

```
            a[j+1] = a[j];
        a[j+1] = x;
    }
}
```

Надо иметь в виду, что индексы массивов в Си всегда начинаются с нуля. Поэтому параметр n обозначает число сортируемых элементов, а номера у них от 0 до n-1. Начальное значение i, равное 1, соответствует второму по порядку элементу в массиве. В условии продолжения внешнего цикла i<n использовано строгое неравенство, поскольку последний элемент в массиве имеет номер n–1. Цикл for в Си — это не цикл с заранее известным числом повторений. В скобках после for записываются три *выражения* (которые могут быть и операторами). Первое вычисляется (выполняется) один раз перед началом цикла, второе задает условие продолжения, а третье вычисляется после каждого витка цикла. Поэтому внутренний цикл в этом варианте программы — не цикл while, как при записи на других языках, а тоже for. Обратите внимание на запись if((x=a[i])<a[i-1]). Здесь видна характерная черта Си — оператор используется как выражение. В первой части условия переменной x присваивается значение a[i], и присвоенное значение используется в сравнении с a[i-1]. Скобки вокруг x=a[i] обязательны, поскольку присваивание имеет меньший приоритет, чем сравнение. Если скобки не поставить, то программа останется синтаксически правильной, но сортировать не будет, поскольку x будет принимать значение 1.0 или 0.0.

Приведу еще один вариант записи функции InsSort, чтобы проиллюстрировать специфические возможности языка Си. Массивы и ссылки в Си неотличимы. В заголовке функции InsSort вместо параметра-массива a[] записан параметр-указатель *a. С помощью операции «,» (запятая) возможно объединение выражений (а значит и операторов). Там, где разрешается выражение, можно записать и несколько выражений, разделенных запятой. Они вычисляются последовательно, а значением всей конструкции становится значение последнего из них. Подобные возможности есть и в Алголе-68. Пользуясь этим, можно объединить во внутреннем цикле сортировки разные виды присваиваний, как предшествующие циклу, так и выполняемые в нем. Обратите также внимание, что изменена инициализация параметра j во внутреннем цикле. Вначале j принимает значение i–1 в индексном выражении, а затем уменьшается еще на единицу.

```
/* Сортировка простыми вставками */
InsSort(float *a, int n) {
   int i, j; float x;
   for( i = 1; i<n; i++ )
      if( (x=a[i])<a[i-1] ) {
         for( a[i]=a[j=i-1], j--;
            j>=0&&x<a[j]; a[j+1]=a[j],j-- );
         a[j+1] = x;
      }
}
```

Точка с запятой после второго оператора `for` принципиально важна. Она изображает пустое тело цикла. Если ее пропустить, то компилятор этого не заметит, но программа работать не будет.

Вряд ли этот вариант, использующий специфические возможности Си, лучше предыдущего. Скорее — наоборот. Он гораздо менее понятен и представляет собой пример плохого стиля.

Стиль Си

Си — популярный язык. Может, даже чересчур популярный. Бывает, его используют для тех задач, для которых он не слишком годится. Этому, по-видимому, способствовало то, что вокруг Си был создан ореол языка профессионального программирования. А кто же не хочет быть профессионалом! Действительно, Си может служить заменой ассемблеру при решении задач системного программирования. Системным программированием, без сомнения, занимаются программисты-профессионалы. Но опытный программист осознает подстерегающие его опасности, у него выработан правильный стиль, он стремится писать простой и понятный код. Новичок же, использующий Си, наоборот, часто считает, что чем больше особенностей языка он использовал, чем короче получился текст программы, чем труднее будет понять использованные им хитроумные приемы другим программистам, тем выше квалификация его самого. Увы, это заблуждение, но язык Си поощряет такой стиль.

Появлению малопонятных, запутанных программ способствуют:

- Выражения в роли операторов и операторы в роли выражений.
- Адресная арифметика.
- Большое число операций и уровней приоритета.
- Условные выражения.
- Операция «,» (запятая).
- Препроцессор.

Необычайно интересной иллюстрацией «возможностей» языка Си является «Международный конкурс запутанных программ на Си» (International Obfuscated C Code Contest), проводимый с 1984 года. С участниками и победителями этого конкурса можно познакомиться на сайте www.ioccc.org. В листинге 1.10 приведена знаменитая программа «Двенадцать дней Рождества» («Twelve Days of Christmas»), участвовавшая в конкурсе 1988 года. Это, без сомнения, выдающийся образец. Ее автор — англичанин Айан Филлиппс (Ian Phillipps).

Листинг 1.10. «Двенадцать дней Рождества»

```
main(t,_,a)char*a;{return!0<t?t<3?main(-79,
-13,a+main(-87,1-_,main(-86, 0, a+1 )+a)):1,
t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==
2 ?_<13 ?main(2,_+1,"%s %d %d\n"):9:16:t<0?
t<-72?main(_,t,"@n'+,#'/*{}w+/w#cdnr/+,{}r\
/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n\
+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e\
#';dq#'l q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]\
'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# ){nl\
]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{\
nl]'/w#q#n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; \
:{nl]'/*{q#'ld;r'}{nlwb!/*de}'c ;;{nl'-{}rw]\
'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/\
 ') }+}{rl#'{n' ')# }'+}##(!!/"):t<-50?_==*a?
putchar(31[a]):main(-65,_,a+1):main((*a=='/')
+t,_,a+1):0<t?main(2,2,"%s"):*a=='/'||main(0,
main(-61,*a,"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\
\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Эта программа написана на «чистом» Си. Самое же удивительное в том, что она печатает. Откомпилируйте и запустите ее без параметров. И вы получите:

On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

On the fourth day of Christmas my true love gave to me
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the fifth day of Christmas my true love gave to me
five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the sixth day of Christmas my true love gave to me
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the seventh day of Christmas my true love gave to me
seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the eighth day of Christmas my true love gave to me
eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the ninth day of Christmas my true love gave to me
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the tenth day of Christmas my true love gave to me
ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the eleventh day of Christmas my true love gave to me
eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the twelfth day of Christmas my true love gave to me
twelve drummers drumming, eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

Впечатляет, не правда ли? Попробуйте проанализировать поведение этой программы. Возможно, у вас прибавится энтузиазма, если вы узнаете, что этим занимались, например, профессионалы из Bell Labs (места рождения Си) и Microsoft Research.

Не могу удержаться, чтоб не упомянуть еще один шуточный сюжет, касающийся языка Си. Уже много лет в компьютерной среде циркулирует датируемый первым апреля документ, озаглавленный «Создатели Си и Unix признают, что разыграли весь мир» (в оригинале по-английски: «CREATORS ADMIT UNIX, C HOAX»). В этом сообщении «всемирно известного информационного агентства» говорится, что Б. Керниган, К. Томпсон и Д. Ритчи признались, что придумали Си и Unix ради шутки, как пародию на Паскаль и операционную систему Multics. «Деннис и Брайан разработали по-настоящему извращенный диалект Паскаля, назвав его «А». Когда мы обнаружили, что другие действительно пытаются писать программы на А, мы быстро добавили еще парочку хитрых примочек, создав В, BCPL, и, наконец, Си. Мы остановились, добившись успешной компиляции следующего:

```
for(;P("\n"),R-;P("|"))for(e=C;e-;P("_"+(*u++/8)%2))
P("| "+(*u/4)%2);
```

Мы не могли даже представить, что современные программисты будут пытаться использовать язык, допускающий подобный оператор!»[18] Шутки шутками, но повод для зубоскальства имеется.

[18] Перевод Д. Кохманюка.

Продолжая серьезный разговор, надо сказать, что, несмотря на существенные по современным представлениям недостатки, на неоправданное внедрение языка в области, для которых он не вполне подходит, Си является одним из важнейших языков программирования. Он послужил прообразом новых языков, став основой целой программистской культуры. Знание и понимание этого языка, безусловно, необходимо каждому профессиональному программисту.

Модульность, надежность, абстракция

В середине 1970-х количество используемых языков стало довольно большим, но ни один из них не удовлетворял в полной мере требованиям, предъявляемым к языку при создании больших программных систем. Фортран не обладал средствами структурного программирования, ПЛ/1 был ненадежен, а трансляторы для него не обеспечивали достаточной эффективности, Паскаль не годился для разработки больших программ, Си не обеспечивал достаточного уровня абстракции.

В этом разделе мы рассмотрим языки Ада и Модула-2, разработанные в конце 1970-х и лишенные перечисленных недостатков. Оба эти языка приблизились к идеалу универсального языка программирования, сформировавшемуся в те годы. И Ада и Модула-2 обеспечивают создание больших и очень больших программ, состоящих из отдельных модулей, которые могут компилироваться раздельно. При этом гарантируется контроль межмодульных связей, что обеспечивает создание надежных программ. Само понятие модуля было сформировано именно этими языками.

Ада

Проблемой унификации языков программирования, необходимостью снизить затраты на разработки и повысить надежность одним из первых озаботилось министерство обороны США — Пентагон. В начале 1975 года была создана рабочая группа, в задачу которой входили выбор или разработка единого языка высокого уровня для систем военного назначения, в первую очередь встроенных систем. Через три месяца после начала работы группа сформулировала предварительный вариант требований к языку, которые получили название «соломенных». Далее, в ходе консультаций с широким кругом специалистов требования уточнялись. Появились «деревянные» (1975),

«оловянные» (1976), «железные» (1978), «стальные» (1979)[19]. Уже на этапе «оловянных» требований стало ясно, что ни один из существующих языков не удовлетворяет им в полной мере. Поэтому в 1977 году был объявлен открытый конкурс на создание нового языка.

Участникам конкурса было рекомендовано взять в качестве прототипа Паскаль, Алгол-68 или ПЛ/1. На конкурс было представлено более полутора десятков проектов. Четыре были отобраны для дальнейшего рассмотрения. Им были даны условные названия Зеленый, Красный, Желтый и Синий. Во всех четырех проектах в качестве основы был выбран Паскаль. В разработке Желтого проекта, представленного компанией SRI International, участвовали Н. Вирт и Ч. Э. Р. Хоар.

После шестимесячного срока, предоставленного авторам для более детальной проработки своих предложений, проекты были вновь оценены с привлечением большого числа экспертов. Для финального этапа конкурса выбрали Зеленый и Красный языки. После еще одного года, отведенного на доработку и тщательный анализ представленных языков, было вынесено окончательное решение. 2 мая 1979 года на заседании рабочей группы Зеленый язык был признан победителем и назван Ада.

Название языку дано в честь Августы Ады Байрон Кинг, графини Лавлейс (Augusta Ada Byron King, Countess of Lovelace), дочери английского поэта Джорджа Байрона. Ада Байрон считается первым в истории программистом. До наших дней дошел документ, в котором Ада дает описание программы вычисления чисел Бернулли для аналитической машины Чарльза Бэббиджа, английского математика и изобретателя XIX века, с которым она сотрудничала.

Язык Ада разработан группой специалистов французской компании CII Honeywell Bull под руководством Жана Ичбиа (Jean D. Ichbiah).

Ада значительно сложнее Паскаля. Автор Паскаля Н. Вирт видел в этом источник серьезных проблем: «Слишком много всего вываливается на программиста. Я не думаю, что, изучив треть Ады, можно нормально работать. Если вы не освоите всех деталей языка, то в

[19] В действительности первый вариант требований назывался «STRAWMAN» — соломенный человек, чучело, но в русской переводческой традиции принято говорить о «соломенных» требованиях. «Чучело» было специально предназначено для избиения и трепки рецензентами. Последующие версии именовались аналогично: WOODENMAN, TINMAN, IRONMAN, STEELMAN.

дальнейшем можете споткнуться на них, и это приведет к неприятным последствиям». На это Ж. Ичбиа возражал: «Иногда Вирт верит в малые решения больших проблем. Я не верю в такие миражи. Крупные проблемы требуют крупных решений». Последний тезис довольно часто выдвигается как оправдание сложности программных систем и языков программирования, хотя, при всей внешней убедительности, не имеет на самом деле достаточных обоснований. «Язык для сложного мира», — такой девиз сопровождает эмблему языка Ада.

Сложность языка Ада таила в себе и другую опасность. Хотя сам язык, предназначенный для использования в чрезвычайно ответственных военных системах, содержал достаточные механизмы контроля, источником ненадежности могли оказаться ошибки в компиляторах, поскольку любой компилятор Ады неизбежно оказывается очень большим. Надо сказать, что эта опасность была осознана своевременно: разработаны строгие процедуры аттестации компиляторов Ады. В 1980 году были сформулированы «каменные» (STONEMAN) требования к программному окружению, которое должно сопровождать каждую реализацию Ады. В 1983 году принят американский ANSI-стандарт языка Ада, а в 1987 году совпадающий с ним международный стандарт ISO.

После принятия стандарта и появления промышленных компиляторов язык Ада стал обязательным для многих военных применений в США, а также в странах НАТО. В гражданской сфере распространенность языка Ада существенно меньше. Он используется в крупномасштабных проектах, связанных с авиацией, транспортом, системами связи.

В нашей стране язык Ада был встречен с энтузиазмом. Достаточно сказать, что русские переводы книг по языку стали выходить уже через два года после его появления (очень оперативно по меркам тех лет). Общедоступные книги по Аде появились даже раньше, чем книги по Паскалю и Си. Видимо, перспективы Ады представлялись нашим аналитикам предпочтительными. Ада был официальным языком «потенциального противника», и уже поэтому заслуживал повышенного внимания.

В отличие от Паскаля и Си, возникших в результате личной инициативы их авторов, язык Ада был создан по государственному плану. На государственную основу поставили и продвижение Ады в СССР. Была образована Рабочая группа по языку Ада при Госкомите-

те по науке и технике. Несколько коллективов разработчиков занимались реализацией Ады. Однако до широкого использования языка, как в военной, так и в гражданской сфере, дело не дошло. Известно, однако, что программное обеспечение авионики (авиационной электроники) на ряде отечественных самолетов (аэробус Ил-96М, летающая лодка Бе-200 КБ Г. М. Бериева) написано на языке Ада. Но это уже события нового времени и речь, по-видимому, идет об импорте западных программных технологий и авионики. Ада используется в программном обеспечении Международной космической станции, построенной совместными усилиями США и России.

Основные черты языка Ада

- Модульное строение программы. Единицами раздельной компиляции являются *подпрограммы*, *модули-пакеты* и *модули-задачи*. Они могут компилироваться как отдельно, так и в составе одного компонента компиляции (файла). При компиляции обеспечивается исчерпывающий контроль межмодульных связей. Модуль может состоять из двух частей: спецификации, описывающей доступную для других модулей информацию, и тела модуля, содержащего детали реализации, которые недоступны другим модулям.
- Средства параллельного программирования. Единицей при параллельном исполнении частей программы является *задача*. Для синхронизации параллельно исполняемых задач предусмотрен механизм «рандеву».
- Настраиваемые (родовые) модули, предусматривающие средства параметризации[20].
- Средства обработки исключений.
- Многообразные типы данных: целые, вещественные, логические, символьные, перечислимые. Подтипы. Предусматривается возможность задания точности и диапазона представления данных.
- Способы организации данных (составные типы): массивы и записи (в том числе с вариантами), строки. В модулях-пакетах могут быть определены приватные типы, детали реализации которых скрыты от пользователя пакета.
- Ссылочные типы (указатели), позволяющие создавать данные динамической структуры.
- Строгий статический контроль соответствия типов.

[20] Аналогичные возможности языка Си++ известны как «шаблоны».

- Полный набор управляющих операторов: **case**, **if**, **while**, **loop**, **for**, **exit**, **return**, **goto**.
- Подпрограммы: процедуры и функции. Параметры подпрограмм разделяются на входные (**in**), выходные (**out**) и изменяемые (**in out**).
- Переопределение (совмещение) операций.
- Стандартизованный ввод-вывод, обеспечиваемый предопределенными пакетами.
- Высокая степень стандартизации. Стандарт не допускает ни подмножеств, ни расширений языка.

«Hello, World!» на Аде

Простейшая законченная программа на Аде представляет собой просто отдельную процедуру.

```
with TEXT_IO;
procedure HELLO is
use TEXT_IO;
begin
   PUT_LINE("Hello, World!");
end HELLO;
```

Предложение **with** указывает, что компилируемый модуль использует стандартный пакет текстового ввода-вывода `TEXT_IO`. Использование предложения **use** обеспечивает «прямую видимость» имен пакета `TEXT_IO`. Для нашего примера это означает, что можно писать просто `PUT_LINE` вместо `TEXT_IO.PUT_LINE`.

В Аде прописные и строчные буквы в идентификаторах не различаются. При этом строчные буквы не входят в набор основных символов языка, а имеют статус дополнительных. И хотя использовать их можно, но по традиции в идентификаторах применяют прописные буквы.

Сортировка вставками на Аде

Как и при описании процедур языка Паскаль, тип параметра-массива в Аде должен быть указан именем типа. Для этого нужно предусмотреть отдельное описание этого типа. Оно должно располагаться внутри какой-либо единицы компиляции, но вне процедуры. Поместим это описание в спецификации пакета `SORT` (листинг 1.11). В этой же спецификации запишем и заголовок процедуры `INSSORT`. Таким образом, пакет `SORT` предоставляет другим модулям тип мас-

сива и процедуру сортировки массивов этого типа. Тип массива определен как неограниченный. Этому типу соответствует любой массив из вещественных (FLOAT) элементов с неотрицательными индексами (NATURAL) произвольного диапазона (**range** <>) .

Листинг 1.11. Пакет для сортировки вещественного массива

```
-- Спецификация пакета SORT
package SORT is
type TARRAY is array (NATURAL range <>) of FLOAT;
procedure INSSORT(A : in out TARRAY);
end SORT;

-- Тело пакета SORT
package body SORT is
procedure INSSORT(A : in out TARRAY) is
I, J : INTEGER;
X    : FLOAT;
begin
   for I in A'FIRST+1..A'LAST loop
      X := A(I);
      if X<A(I-1) then -- Сокращаем число пересылок
         A(I) := A(I-1);
         J := I - 1;
         while J>=A'FIRST and then X<A(J) loop
            A(J+1) := A(J);
            J := J - 1;
         end loop;
         A(J+1) := X;
      end if;
   end loop;
end INSSORT;
end SORT;
```

Сама процедура сортировки определяется в теле (**package body**) пакета SORT. На ее примере можно увидеть некоторые особенности языка Ада.

Комментарии в программе оформляются с помощью символов «--» (повторенный дважды «минус»). Часть строки, следующая за этими знаками, считается примечанием. Для обращения к границам индексов массива использованы атрибуты массива A'FIRST (индекс первого элемента) и A'LAST (индекс последнего элемента). Все управляющие операторы завершаются явно: **if** – **end if**, **loop** – **end loop**. Для записи индексов используются круглые скобки. Такая запись,

совпадающая по форме с вызовом функции, вероятно, должна была подчеркнуть, что получение элемента массива подобно вычислению значения функции, аргументами которой являются индексы. Обратите внимание на конструкцию **`and then`** в условии цикла **`while`**. Такая форма записи логического «и» определяет короткую схему вычисления логического выражения. Если в этом примере применить просто **`and`**, будет работать полная схема, и при выполнении программы индекс выйдет за границы массива. Аналогично для задания короткой схемы при использовании «или» применяется форма **`or else`**.

Ада 95

В 1995 году принят обновленный стандарт (одновременно ANSI и ISO) на язык Ада. Определяемый этим стандартом язык называют Ада 95, а язык, соответствующий прежнему стандарту, — Ада 83. Главное нововведение — средства объектно-ориентированного программирования, обеспечивающие наследование и динамический полиморфизм: классы, производные и помеченные типы, расширение записей, абстрактные типы и подпрограммы, динамический отбор подпрограмм (процедурный тип). Кроме того предусмотрена иерархическая организация модулей в библиотеках, определен интерфейс с другими языками программирования, в частности с Фортраном, Коболом и Си. В приложениях к стандарту определены средства, поддержки таких областей применения как системное программирование, системы реального времени, распределенные и информационные системы, численные применения, системы, критичные с точки зрения безопасности.

Модула-2

Язык Модула-2 разработан в ходе реализации проекта рабочей станции Lilith. Этот проект, инициированный, вдохновляемый и руководимый автором языка Паскаль Никлаусом Виртом, был запущен в ETH (Цюрих, Швейцария) в конце 1977 года. Основной идеей была совместная разработка аппаратуры и программного обеспечения, свободная от каких-либо ограничений, требующих совместимости с существующими аппаратными и программными платформами. Компьютер проектировался таким образом, чтобы максимально упростить компиляцию программ с языка высокого уровня, обеспечивая при этом высокую эффективность. Идея подчинения компьютерной архитектуры потребностям трансляции с языков высокого уровня,

а именно на таких языках пишется подавляющее число программ, — один из основных тезисов, пропагандируемых Н. Виртом.

Все программы планировалось написать на одном языке. Но ни один из существующих языков Вирта не устраивал. Язык должен был поддерживать, подобно Паскалю, хорошие возможности структурирования, обеспечивать высокий уровень абстракции, гарантировать надежность, и в то же время предоставлять средства низкого уровня для непосредственного доступа к аппаратуре. Кроме этого, язык должен быть достаточно прост, чтобы разработка компилятора (конечно же, на самом этом языке) могла быть выполнена за разумное время, а сомнений в правильной работе этого компилятора не возникало.

Предложенный Н. Виртом язык был назван Модула-2. Название подчеркивает основную его особенность — наличие модулей — раздельно компилируемых частей программы со строгим интерфейсом, определяющим их взаимодействие. Модула-2 — потомок Паскаля и экспериментального языка Модула, использовавшегося Н. Виртом для исследований по мультипрограммированию. Значительное влияние на Модулу-2 оказал также язык Mesa, с которым Н. Вирт познакомился в исследовательском центре компании Xerox в Пало-Альто (PARC), где он провел свободный от лекций год (1976).

Основная работа по спецификации и реализации языка происходила в 1978-79 годах. Первый компилятор Модулы-2 заработал в 1979 году на мини-компьютере PDP-11. С его помощью велась разработка программного обеспечения для Lilith. В дальнейшем он подвергся нескольким модернизациям.

Описание нового языка было опубликовано в 1980 году. Позднее, в 1983 году, в язык были внесены изменения. В 1984 Н. Вирт разработал новый компилятор Модулы-2 для Lilith, благодаря однопроходной схеме трансляции работавший в пять раз быстрее прежнего. Благодаря простоте языка размер компилятора составил всего около 5 тысяч строк.

Новый язык быстро стал известным благодаря большому интересу к работам Н. Вирта, который к тому времени приобрел значительный авторитет в компьютерном мире. Появились коммерческие компиляторы Модулы-2 для различных платформ. Первым был компилятор фирмы Logitech. На IBM PC-совместимых компьютерах получил известность компилятор компании TopSpeed, образованной бывшими сотрудниками фирмы Borland. В корпорации IBM Модулу-2 использовали для программирования операционной системы компьютеров AS/400. В 1996 году был принят стандарт ISO на язык Модула-2.

Несмотря на свои очевидные достоинства, язык Модула-2 не получил слишком большого распространения. Этому можно найти несколько объяснений.

Во-первых, в 1983 году появился Turbo Pascal — выдающаяся в техническом отношении система, обладавшая высокой скоростью работы, компактностью и удобством в использовании. Успех первой версии позволил компании Borland вложить достаточные средства в развитие и продвижение Turbo Pascal. Большое число потенциальных пользователей Модулы-2, были привлечены бурно развивавшимся и активно продвигаемым Turbo Pascal. Это при том, что язык Turbo Pascal безусловно уступал Модуле-2 по качеству реализации механизма модульного программирования и многим другим критериям.

С другой стороны, в 1980-е годы продолжилось распространение операционной системы Unix, основным языком которой был Си. Язык Си и последовавший за ним Си++ были поддержаны крупными разработчиками коммерческого программного обеспечения. Потерпев поражение в конкурентной борьбе с Borland на рынке компиляторов языка Паскаль, корпорация Microsoft сосредоточила свои усилия на продвижении средств разработки на языках Си и Си++ (наряду с Visual Basic). Си и Си++ стали и основными языками операционной системы Windows. В свою очередь, Borland также уделила немалое внимание Си и Си++, поскольку не могла ограничиться в своих системах языком, который не соответствовал стандартам и не применялся за рамками этих систем.

Судьба языка Модула-2 в нашей стране аналогична его мировой судьбе — высокий интерес вначале и не слишком большое распространение в дальнейшем. Перевод книги Н. Вирта «Программирование на языке Модула-2» был выпущен почти сразу вслед за книгами по Паскалю. Модула-2 была реализована на отечественных компьютерах, в частности, в системе Эльбрус[21]. На компьютерах с системой

[21] К середине 1980-х подавляющее число выпускаемых в СССР компьютеров представляли собой клоны моделей американских фирм IBM, DEC, Hewlett Packard, Wang. «Заимствовалось» и программное обеспечение. Серия вычислительных комплексов Эльбрус была одной из немногих оригинальных разработок. Один из ее основных принципов — ориентация на языки высокого уровня. Роль машинного языка Эльбруса выполнял язык высокого уровня Эль-76. Язык ассемблера вообще отсутствовал.

команд PDP-11 (СМ-4, Электроника-60, ДВК) получили некоторое распространение компиляторы ETH.

Отдельно следует отметить тесно связанный с языком Модула-2 проект «Кронос». Начиная с 1983 года группой студентов Новосибирского университета (Е. Тарасов, Д. Кузнецов, А. Недоря, В. Васекин, руководитель чл.-корр. АН СССР В. Е. Котов), впоследствии — сотрудников ВЦ СО АН СССР, образовавших Cronos Research Group, была спроектирована, изготовлена, оснащена операционной системой, компиляторами и прикладными программами 32-разрядная рабочая станция Кронос. Основным языком, на котором разрабатывались программы для Кроноса, был Модула-2. В основе проекта лежали примерно те же идеи, что и в проекте Lilith. Именно знакомство с Модулой-2 и Lilith и послужило толчком к началу работы. Компьютер Кронос продемонстрировал эффективность при решении ряда задач, было изготовлено несколько сотен его экземпляров. В 1985-90 годах технология Кронос стала одной из ключевых в советском проекте МАРС, целью которого было создание высокопроизводительных и высокоинтеллектуальных отечественных вычислительных систем. Тем не менее, крупномасштабный проект МАРС так и не получил практического воплощения.

Опыт, приобретенный в проектах Кронос и МАРС, не был утрачен. Образованная в Новосибирске участниками Cronos Research Group во главе с А. Е. Недорей компания xTech Ltd. занялась разработкой систем программирования для языков Модула-2 и Оберон-2 на основе собственной технологии, названной XDS и обеспечивающей перенос систем на многие аппаратно-программные платформы. Однако в 2000-е годы разработка систем программирования семейства XDS для языка Модула-2 и Оберон-2 была прекращена.

Основные черты языка Модула-2

- Раздельно компилируемые модули. В программе могут быть главный модуль, модули реализации и модули определений, а также локальные (вложенные внутрь других) модули. Одни модули могут импортировать средства, экспортированные другими. Модуль определений содержит перечень экспортируемых констант, типов, переменных, процедур. Реализация элементов, записанных в модуле определений, обеспечивается соответствующим модулем реализации. Модуль определений и модуль реализации компилируются

отдельно. Предусмотрен только явный импорт идентификаторов, позволяющий однозначно определить из какого модуля импортируется тот или иной идентификатор.

- Усовершенствованный по сравнению с Паскалем синтаксис управляющих операторов. Отсутствуют составной оператор и оператор перехода. Большинство управляющих операторов (кроме **REPEAT** – **UNTIL**) завершаются явно: **IF** – **THEN** – **ELSIF** – **ELSE** – **END**, **CASE** – **OF** – **ELSE** – **END**, **WHILE** – **DO** – **END**, **FOR** – **TO** – **BY** – **DO** – **END**, **LOOP** – **END**. Операторы выхода (из цикла **LOOP**) **EXIT** и возврата (из процедуры) **RETURN**.
- Основные типы: INTEGER — целые со знаком, CARDINAL — неотрицательные целые, BOOLEAN, CHAR, REAL и LONGREAL, LONGINT.
- Перечислимый и ограниченный типы, массивы, записи и множества. Тип BITSET — множество целых, представленное одним машинным словом. Указатели.
- Процедурный тип.
- Открытые массивы (без указания длины), которые могут быть параметрами процедур.
- Жесткий статический контроль типов. Смешение данных разных типов в одном выражении и неявные преобразования запрещены.
- Средства мультипрограммирования (параллельного программирования): процессы, мониторы, сигналы.
- Средства программирования низкого уровня, локализованные в отдельном (псевдо)модуле SYSTEM. Упоминание SYSTEM в списке импорта некоторого модуля означает, что этот модуль является машинно-зависимым и потенциально небезопасным.
- Отсутствие в языке средств ввода-вывода. Ввод-вывод обеспечивается набором модулей, сопровождающих каждую реализацию. Обычно системы программирования на Модуле-2 содержали модули, описанные в книге Н. Вирта «Programming in Modula-2». Стандарт ISO определял свой набор стандартных модулей.

Некоторые решения, принятые в Модуле-2, Н. Вирт впоследствии оценивал как не очень удачные или вынужденные. Это относится к возможности использовать имя типа как название функции, интерпретирующей данные одного типа как имеющие тип, обозначенный функцией (приведение типов). Эта необходимая, но машинно-зависимая и опасная возможность оказалась нелокализованной, и части программы, где она используется, оказываются никак не помечены, в отличие от случая импорта средств низкого уровня из модуля SYSTEM. Аналогич-

ную проблему создают и записи с вариантами. В последовавшем за Модулой-2 языке Оберон эти изъяны были исправлены.

Примеры программ на Модуле-2

Простейшая законченная программа на Модуле-2 состоит из одного главного модуля. Строчные и заглавные буквы в идентификаторах Модулы-2 различаются. Ключевые слова всегда записываются заглавными буквами.

```
MODULE Hello;
IMPORT InOut; (* Простая форма импорта *)
BEGIN
   InOut.WriteString("Hello, World!");
   InOut.WriteLn
END Hello.
```

Процедуры вывода строки (`WriteString`) и перехода на новую строку (`WriteLn`) импортируются из стандартного модуля `InOut`. В списке импорта упоминается только название этого модуля. Импортируемые идентификаторы в тексте программы каждый раз сопровождаются (уточняются, квалифицируются) именем модуля. Это универсальный способ импорта, но при частом употреблении импортируемых идентификаторов запись может получиться довольно громоздкой. Предусмотрен другой вариант оформления импорта, при использовании которого наша программа будет выглядеть так:

```
MODULE Hello;
FROM InOut IMPORT WriteString, WriteLn;
BEGIN
   WriteString("Hello, World!");
   WriteLn
END Hello.
```

В этом случае тоже обеспечивается однозначное определения модуля, из которого импортируется каждый идентификатор. Но, в отличие от первого варианта, невозможен одновременный импорт одинаковых идентификаторов из двух разных модулей. В дальнейшем, в языке Оберон Н. Вирт оставил лишь первый вариант импорта, чуть усовершенствовав его.

Перейдем к сортировке. Модула-2, в отличие от Паскаля, позволяет не указывать в параметрах процедуры конкретное имя типа для массива. Достаточно указать тип элементов массива. Это делает такую процедуру гораздо более универсальной, позволяя с ее помо-

щью обрабатывать массивы, имеющие любые диапазоны индексов. Определенные таким образом параметры-массивы называют открытыми или гибкими. Внутри процедуры открытый массив a считается проиндексированным значениями от 0 до HIGH(a). Стандартная функция HIGH дает значение на единицу меньшее числа элементов массива.

```
(* Сортировка вставками на Модуле-2 *)
PROCEDURE InsSort(VAR a: ARRAY OF REAL);
VAR
   i, j : CARDINAL;
   x    : REAL;
BEGIN
   FOR i := 1 TO HIGH(a) DO
      x := a[i];
      IF x<a[i-1] THEN
         a[i] := a[i-1];
         j := i;
         WHILE (j>0) AND (x<a[j-1]) DO
            DEC(j);
            a[j+1] := a[j];
         END;
         a[j] := x;
      END;
   END;
END InsSort;
```

Использование в роли индексов переменных i и j типа CARDINAL требует внимательного программирования внутреннего цикла. Записывать в условии цикла **WHILE** (j>=0) … уже нельзя. Отрицательное число просто не может быть значением j, и выражение j>=0 всегда истинно. Поэтому в условии цикла используется строгое неравенство j>0, изменение j происходит в начале, а не в конце цикла. Для уменьшения j на единицу вместо оператора j := j – 1 использован вызов стандартной процедуры DEC. Это помогает компилятору сформировать более эффективный машинный код. Составные условия всегда вычисляются по короткой схеме[22].

---

[22] Короткая схема вычисления составных условий предполагает, что в выражениях вида (j>0) **AND** (x<a[j-1]) при невыполнении первой части условия (то есть при j<=0) вторая часть (x<a[j-1]) не проверяется, поскольку и без этого понятно, что условие ложно.

Абстрактные типы данных

Тип данных — это совокупность множества значений и множества операций.

Если речь идет о стандартном типе, встроенном в язык программирования, то и набор значений и набор операций заданы описанием языка. При этом языки высокого уровня, как правило, не регламентируют внутреннее представление того или иного типа данных. Например, для использования вещественных чисел программисту совсем не обязательно знать, каким способом эти числа представлены в памяти. Представление данных скрыто от программиста. Доступ к данным осуществляется через набор допустимых операций.

При конструировании собственных типов и структур данных также разумно придерживаться подобного подхода, отделяя внутреннее представление от набора операций. Более того, говоря о той или иной структуре данных, имеет смысл определять ее через набор применимых операций, абстрагируясь от способа хранения данных в этой структуре.

Такой подход, предусматривающий отделение спецификации операций, применимых к данным, от внутреннего представления этих данных и реализации операций и составляет концепцию абстрактных типов данных (АТД).

Программист, использующий абстрактный тип данных, может не интересоваться (и даже *не должен* интересоваться) внутренней организацией данных. Ему достаточно иметь в распоряжении набор операций, выраженных с помощью процедур и функций. Реализация этих процедур и функций также должна быть скрыта от использующего абстрактный тип данных.

Модули в языках Ада и Модула-2 предоставляют необходимые средства для реализации абстрактных типов данных. Использование модулей для воплощения абстрактных типов данных — это и есть один из основных вариантов их применения.

Рассмотрим пример, выбрав на роль абстрактного типа данных очередь.

Очередью назовем структуру (тип) данных, для которой определены:

1. Тип элементов, из которых состоит очередь (обозначим этот тип `tData`).
2. Операция создания (инициализации) очереди (процедура `Init`).

3. Операции добавления (`Put`) элементов в очередь и извлечения (`Get`) элементов из очереди. Операции добавления и удаления действуют таким образом, что элемент, добавленный первым, первым и извлекается. Такую дисциплину добавления и исключения элементов обозначают FIFO (First In First Out — «первым пришел, первым ушел»).
4. Операция, проверяющая наличие элементов в очереди (функция `NotEmpty`).

Как видите, в этих определениях отсутствуют сведения о том, как представлена очередь. Чем она является «внутри». Массивом? Списком? Файлом? Это не важно для того, чтоб называть очередь очередью. Мы определили ее через операции, которые с нею можно выполнять. Это и есть взгляд на очередь, как на абстрактный тип данных.

Запишем на языке Модула-2 модуль определений, предоставляющий в распоряжение программиста абстрактный тип данных очередь.

```
(* Очередь *)
DEFINITION MODULE Queue;

FROM Data IMPORT tData;

TYPE tQueue;

(* Создать очередь Q *)
PROCEDURE Init(VAR Q: tQueue);

(* Добавить элемент D в очередь Q *)
PROCEDURE Put(VAR Q: tQueue; D: tData);

(* Получить в D элемент из очереди Q *)
PROCEDURE Get(VAR Q: tQueue; VAR D: tData);

(* Очередь Q не пуста *)
PROCEDURE NotEmpty(Q: tQueue): BOOLEAN;

END Queue.
```

Чтобы меньше привязывать модуль, определяющий очередь, к конкретному типу элементов, из которых она может состоять, определение типа элементов вынесено в отдельный модуль `Data`. Этот модуль импортируется нашим модулем `Queue` и должен также импортиро-

ваться модулем, использующим очередь. Модуль определений Data может, например, быть таким:

```
DEFINITION MODULE Data;
TYPE tData = REAL;
END Data.
```

При изменении типа данных, составляющих очередь, модуль Queue не меняется, но должен быть откомпилирован заново.

Язык Модула-2 предоставляет возможность скрытого экспорта типов. Это соответствует принципам абстракции данных, позволяя прятать внутреннее представление абстрактного типа. В нашем примере таким способом модуль Queue экспортирует тип tQueue. В модуле определений упомянуто лишь имя этого типа. Правда, Модула-2 предусматривает скрытый экспорт лишь типов-указателей, поэтому, раскрывая структуру типа tQueue в модуле реализации, мы должны будем написать:

```
TYPE tQueue = POINTER TO ...
```

Это, однако, не слишком сужает возможности реализации, поскольку тип tQueue может быть и указателем на массив и указателем на запись. Но переменные нашего абстрактного типа все же смогут существовать лишь в динамической форме. Альтернативой могло быть раскрытие внутренней структуры типа tQueue в модуле определений, что провоцирует программиста на использование прямого доступа к элементам структуры в обход предусмотренного набора процедур-операций. Ну а это разрушает абстракцию данных, делает части программы, использующие очередь, зависимыми от ее реализации, не позволяет изменить реализацию без изменений в модулях, использующих очередь.

Объектно-ориентированное программирование

Технология и идеология объектно-ориентированного программирования (ООП) зародились в 1960-е годы, в период активного осмысления технологических вопросов разработки программных систем. Свое воплощение тогда ООП получило в языке Симула-67, созданном Оле-Йоханом Далом (Ole-Johan Dahl) и Кристеном Нюгардом (Kristen Nygaard) в Норвежском компьютерном центре. Симула-67 предназначался для программирования задач имитационного моделирования и представлял собой модернизированный Алгол-60.

Можно считать, что объектно-ориентированное программирование — это раздел структурного программирования, ведь одна из основополагающих работ по ООП «Иерархические структуры программ» О.-Й. Дала и Ч. Э. Р. Хоара была опубликована под одной обложкой со статьей Э. Дейкстры «Заметки по структурному программированию» в книге, которая так и называлась «Структурное программирование».

В те годы предложенный Симулой-67 подход к построению программ не получил массового распространения. Новая волна интереса к ООП возникла в 80-е годы в связи с появлением систем с графическим пользовательским интерфейсом, при создании которых использование ООП оказалось весьма уместным.

Что такое объектно-ориентированное программирование

Можно выделить два уровня объектно-ориентированного программирования. Во-первых, это особый подход к разработке программ, взгляд на программные компоненты как на объекты, то есть определенная, «объектная» манера мышления, «объектная идеология». Во-вторых, с ООП связывают ряд конкретных механизмов, реализованных в объектно-ориентированных языках, и составляющих технику объектно-ориентированного программирования.

Идеология ООП

Традиционный взгляд на программу — ее представление как совокупности *действий*, реализуемых отдельными частями программы — процедурами. Естественный способ разработки при этом — пошаговая детализация, понимаемая как разбиение основной задачи на более мелкие действия-подзадачи с последующей детализацией этих действий. В ходе разработки проектируются и уточняются и структуры данных, но ведущую роль все же выполняет декомпозиция *действий*. Это процедурное программирование.

Основной принцип объектно-ориентированного программирования — представление программы как совокупности *объектов*. Может быть точнее, представление *о программе*, как *о совокупности* объектов. В роли объектов выступают переменные-записи (как правило, динамические), каждая из которых хранит данные, определяющие текущее *состояние* объекта. С каждым *типом объектов* связан набор действий, который объекты могут выполнять и которые определяют *поведение* этого типа объектов. Эти действия-процедуры могут обра-

щаться к данным, определяющим состояние объекта, и могут менять эти данные. Выполнение процедуры, одним из параметров которой (первым параметром) является переменная-объект, трактуется не как выполнение действия *над* этой переменной, а как исполнение *объектом* некоторого действия в ответ на поступившее *сообщение* — вызов процедуры.

Объектный подход предполагает следующие изменения в организации и разработке программ.

- Возрастание роли данных, воспринимаемых как состояние объектов.
- Децентрализация управления в программе. Основную работу выполняют процедуры нижнего уровня, связанные с объектами различных типов и реализующие поведение этих объектов. Вызовы связанных с типом объекта процедур трактуются как передача сообщений объекту.
- Доступ извне к состоянию объекта, ограничивается или запрещается. Взаимодействие с объектом осуществляется исключительно (или почти исключительно) путем передачи сообщений (вызовов процедур).
- Возрастание роли восходящих методов разработки (в противоположность нисходящей пошаговой детализации), когда вначале проектируется совокупность (типов) объектов, а затем организуется их взаимодействие.

Существует мнение, и с ним трудно не согласиться, что более точным названием обсуждаемой технологии было бы «субъектно-ориентированнное программирование», поскольку речь идет о придании большей активности частям программы, воспринимаемым как *субъекты*, обладающие самостоятельным поведением. Объектами же манипулируют извне. Будем, однако, придерживаться устоявшейся терминологии.

Пример объектного подхода

Подобно тому, как на языке, не содержащем достаточного набора структурных операторов (например, на Фортране ранних версий или на языке ассемблера), можно программировать в структурном стиле, используя переходы только как средство реализации стандартных управляющих структур, на языке, не содержащем специальных конструкций, поддерживающих ООП, можно программировать в объектном духе.

Вот пример из моей практики. В графическом редакторе Турбограф (рис. 1.15) имеется индикатор, расположенный в правом нижнем углу рабочего поля. Этот индикатор (показан в увеличенном виде на рис. 1.15) по ходу рисования должен отображать либо текущие координаты курсора мыши, либо координаты угла и размеры прямоугольника, который вы наносите на рисунок, либо координаты центра окружности и ее радиус и т. д.

Рис. 1.15. Графический редактор Турбограф с индикатором в правом нижнем углу

Первая попытка реализации этого индикатора выглядела так. В той части программы, которая отвечала за перемещение и отображение курсора мыши, было написано (Турбограф программировался на Турбо Паскале) примерно следующее:

```
GetMouseXY(Mx, My); {Получить координаты мыши}
if (Mx<>OldMy) or (My<>OldMy) then begin
   { Мышь передвинута }
   ClearIndicator;                    {Очистить индикатор}
    Str( Mx, Sx); Str(My, Sy);        {Преобразования в стро-
ку}
   SetColor(DigitColor);              {Цвет цифр на индикато-
ре}
   OutTextXY(IndX1, IndY1, Sx); {Вывести Mx}
   OutTextXY(IndX2, IndY2, Sy); {Вывести My}
   OldMx := Mx; OldMy := My;
   ...
```

Однако сразу стало ясно, что это плохой вариант. Отображение данных на индикаторе должно занимать минимум времени. Основные ресурсы должны расходоваться на рисование в рабочем поле. А если при любом перемещении мыши индикатор целиком очищается (закраской прямоугольника), а затем выводятся два числа, даже если изменилось только одно из них, то затраты времени на такую работу получаются слишком большими[23]. К тому же гарантировано максимальное мерцание индикатора. Тогда можно попробовать написать что-то такое:

```
GetMouseXY(Mx, My); {Получить координаты мыши}
if Mx<>OldMx then begin
   Str(Mx, S);
   Str(OldMx, OldS);            {Преобразования в строку}
   SetColor(BgColor);           {Цвет поля индикатора}
   OutTextXY(IndX1, IndY1, OldS); {Стереть старое Mx}
   SetColor(DigitColor);          {Цвет цифр          }
   OutTextXY(IndX1, IndY1, S);    {Вывести новое Mx  }
   OldMx := Mx;
   ...
end;
if My<>OldMy then begin
   Str(My, S);
   Str(OldMy, OldS);            {Преобразования в строку}
   SetColor(BgColor);           {Цвет поля индикатора}
   OutTextXY(IndX2, IndY2, OldS); {Стереть старое My}
```

[23] Для компьютера IBM PC/XT, попавшего в СССР в 1991 году, это и вправду были немалые затраты.

```
    OutTextXY(IndX2, IndY2, S);     {Вывести новое  My}
    OldMy := My;
    ...
 end;
```

С точки зрения быстродействия этот вариант лучше. Но ведь кроме режима перемещения курсора мыши есть масса других вариантов работы индикатора. И каждый раз программировать такой громоздкий фрагмент? А как быть при смене режима рисования, когда, к примеру, взамен четырех чисел надо выводить два или три? А если предусмотреть отключение индикатора, то в каждый такой фрагмент надо добавлять еще одну проверку.

И тогда был применен объектный подход. При этом не использовались специальные механизмы ООП, имевшиеся в Turbo Pascal версии 6.0. Фрагмент программы, обеспечивающий работу индикатора, был просто частью модуля, отвечавшего за отображение элементов управления Турбографа. Этот фрагмент включал:

- Совокупность переменных, определяющих текущее состояние индикатора: режим его работы (отображение двух координат; координат, ширины и высоты; координат и радиуса; исходный режим), включен или выключен индикатор, текущие отображаемые значения, координаты индикатора на экране, координаты чисел, выводимых на индикатор. Эти переменные, являясь глобальными внутри своего модуля, недоступны из других модулей.
- Процедуры, с помощью которых на индикатор выводятся нужные данные:

```
 {Вывести две координаты}
    procedure ShowXY(X, Y: integer);
 {Вывести координаты, ширину и высоту}
    procedure ShowXYWH(X, Y, W, H: integer);
 {Вывести координаты и радиус}
    procedure ShowXYR(X, Y, R: integer);
```

Эти процедуры (и только они) доступны из других модулей. Устроены они довольно сложно, поскольку при каждом своем вызове полностью анализируют ситуацию, используя значения переменных состояния и полученные параметры. Определяют, включен ли индикатор. Если включен, то совпадает ли его режим с тем, что нужен данной процедуре. Если совпадает, то равны ли значения параметров процедуры текущим отображаемым значениям. В зависимости от результатов таких проверок индикатор либо очищается полностью, а затем вы-

водятся нужные значения, либо стираются только несовпадающие значения, и выводятся измененные, а может и ничего не происходить. Большое число проверок не снизило скорость работы, поскольку, как показали наблюдения, затраты на эти проверки несоизмеримо меньше времени, необходимого на само рисование.

Теперь индикатор мог восприниматься как объект, которому достаточно передать сообщение о необходимости показать данные, а он сам решает, как это сделать. Вызывающая программа при этом резко упрощается. Можно даже не проверять, изменилось ли положение мыши:

```
GetMouseXY(Mx, My); {Получить координаты мыши}
ShowXY(Mx, My);     {Показать на индикаторе}
```

Если нужно вывести координаты и радиус, достаточно написать:

```
ShowXYR(Mx, My, R);
```

При этом совершенно не нужно беспокоиться о том, что было на индикаторе до этого.

Как видите, можно мыслить и программировать в объектном стиле, используя вполне традиционные средства. Надо, однако, заметить, что рассмотренный нами пример с индикатором демонстрирует лишь частную ситуацию. Индикатор присутствует на экране и как часть программы в единственном числе, в то время как при разговоре об объектах и ООП важно рассмотреть случай, когда объекты существуют во многих экземплярах.

Расширение типов

Нетрудно заметить схожесть объектно-ориентированного программирования и концепции абстрактных типов данных (АТД). Действительно, объект — это не что иное, как переменная абстрактного типа. Общим является и принцип упрятывания внутреннего представления данных, и ограничение доступа к данным с помощью предопределенного набора операций.

Принципиальным отличием, делающим ООП богаче АТД, является возможность расширения типов. Один тип может быть определен на основе другого, обогащая и модифицируя своего предшественника. В. Ш. Кауфман [Кауфман, 1993] даже использует термин «обогащение типов» вместо понятия «расширение типов». Один из апологетов ООП Г. Буч предлагает называть «объектными» языки, поддерживающие абстракцию данных, а «объектно-ориентированными» —

языки, допускающие расширение типов [Буч, 1999]. К первым можно отнести Модула-2, Ада-83, ко вторым — Си++, Оберон, Ада-95, Ява, Си#.

Наличие механизма расширения типов важно вот по какой причине. Во многих задачах, где уместен объектный подход, можно выделить *родственные* типы объектов. Объекты родственных типов обладают общими чертами в поведении, могут выполнять однотипные операции, но выполнять их по-разному. Например, если в качестве объектов рассматриваются элементы графического интерфейса (окна, кнопки, поля ввода и т. п.), то каждый из таких элементов должен «уметь» себя нарисовать в ответ на получение соответствующего сообщения (вызов процедуры), должен реагировать на щелчок мышью. Но каждый такой объект (точнее, тип объекта) рисует себя и реагирует на щелчок мыши по-своему. То есть виды (и названия) выполняемых действий (процедур) одни, но содержание их разное. Типы объектов с родственным поведением и устройством можно определять как иерархию, в которой типы, обладающие общим свойством в строении и поведении, являются расширением одного типа, обладающего этими свойствами, который как бы является их общим *предком*, а они — его *наследниками*. Например, типы объектов, соответствующие окну, кнопке, полю ввода могут быть расширением типа «реагирующий на щелчок мыши», а тот в свою очередь — расширением типа «умеющий себя нарисовать».

Из объектов можно образовывать динамические структуры, например, списки, состоящие из разнотипных, но родственных элементов. Благодаря тому, что типы этих объектов образуют иерархию и совместимы в рамках этой иерархии, удается обеспечить достаточный уровень статического (во время компиляции) и динамического (во время выполнения) контроля при оперировании таким списком. Например, если окно содержит несколько элементов диалога (полей ввода, кнопок и т. п.), то эти элементы-объекты могут храниться в списке, связанном с объектом-окном. При проектировании такого списка можно предусмотреть, чтобы он состоял из объектов типа «умеющий себя нарисовать» или наследников этого типа. При передаче по такому списку сообщения «всем нарисоваться!» можно быть уверенным, что все объекты, образующие список, смогут это предписание исполнить. Других в список просто не пускали. Но, конечно, кнопка и поле ввода нарисуют себя по-разному.

Терминология и техника ООП

До сих пор в разговоре об ООП мы в основном обходились традиционными терминами, используя слова «объект» и «передача сообщения» вместо «переменная» и «вызов процедуры» лишь как метафоры, помогающие по-новому взглянуть на строение программы. Вместе с тем во многих объектно-ориентированных языках существует и специальная терминология.

Говорят, что объекты-переменные одного типа образуют *класс*. То есть класс — это тип объектов. Объекты в этом случае называются также экземплярами класса. Связанные с классом процедуры и функции обычно называют *методами*, точнее методами экземпляра или функциями-членами. Расширение типов — *наследование*. Рассматриваются классы-предки (*суперклассы*) и классы-потомки (*субклассы, производные классы*).

Справедливости ради надо отметить, что понятие «класс» появилось в программировании не только не позже, но даже раньше понятия «тип запись». Термин «класс» введен в обиход языком Симула-67. Это новое понятие стало обобщением понятия «блок» Алгола-60. Блок представляет собой совокупность описаний (переменных, процедур) и операторов. В языках, происходящих от Си и Симулы-67, классы — это такие блоки, для которых могут быть созданы экземпляры-объекты. В объектно-ориентированных языках, восходящих к Паскалю, объекты — это переменные такого типа запись, частью которого являются процедуры.

Средства объектно-ориентированных языков, обслуживающие наследование, составляют технику ООП. Предусматриваются специальные формы записи, позволяющие указать, что один тип (класс) является расширением (подклассом, производным классом) другого.

Описания методов (или их заголовки) могут помещаться внутри описания класса (типа запись). Рассмотрение и оформление методов как части описания типа (класса) называют *инкапсуляцией*. Некоторые авторы этим же словом обозначают и принцип разделения спецификации поведения объекта и описания его внутреннего устройства [Буч, 1999].

Совместимость объектов в рамках иерархии классов и способность разнотипных, но имеющих общего предка объектов по-своему реагировать на одно и то же сообщение обозначаются термином *полиморфизм*. Полиморфизм реализуется *виртуальными методами* (*вирту-*

альными функциями) — связанными с типом объектов процедурами и функциями, которые могут замещать (перекрывать) друг друга. Виртуальный метод класса-предка может быть подменен (перекрыт) одноименным методом класса-потомка. Какой из одноименных виртуальных методов будет вызван, определяется лишь на стадии выполнения программы. В этом случае говорят о *позднем связывании*.

Ряд специальных конструкций объектно-ориентированных языков обеспечивают проверку фактического типа объектов и преобразование (приведение) типов в рамках иерархии наследования.

Пример использования механизмов инкапсуляции, наследования и полиморфизма мы рассмотрим после знакомства с языками объектно-ориентированного программирования.

Лекарство от всех болезней

Распространение идей объектно-ориентированного программирования совпало по времени с появлением и массовым распространением персональных компьютеров. Развитие программного обеспечения для компьютеров массового применения, в частности, создание графического интерфейса пользователя, способствовало внедрению объектных походов.

Однако резкое расширение круга людей, имеющих отношение к компьютерным технологиям, привело и к нежелательным эффектам. Пропаганда технологических идей в столь массовой аудитории приобрела черты коммерческой рекламы товаров массового спроса. Объектно-ориентированное программирование преподносилось как универсальный рецепт решения проблем, возникающих при разработке больших систем программного обеспечения, как чуть ли не единственно возможный подход к созданию программ. Это не могло не нанести ущерб, в том числе и самой идее объектно-ориентированного программирования.

Универсальность ООП в общем-то не вызывает сомнений. Действительно, если задаться такой целью, то любую задачу можно запрограммировать, используя объекты и ничего кроме объектов. Этот тезис подтверждается, например, существованием языка Смолток (Smalltalk), в котором все является объектами. Даже число — это объект, который может, например, сложить себя с другим объектом, получив соответствующее сообщение.

Универсальность, однако, не означает, что применение ООП к любой ситуации является оправданным. В тех задачах, где действительно присутствуют элементы, которые представляют некоторые замкнутые образования, имеют определенное физическое (графическое) представление, обладают сложным внутренним устройством и могут быть наделены собственной активностью, объектный подход, безусловно, уместен. К таким задачам можно отнести в первую очередь имитационное моделирование (из которого и возникло ООП) и построение графического пользовательского интерфейса. Характерной чертой этих и подобных задач является существование в ходе исполнения программы большого числа родственных по природе и зачастую короткоживущих элементов, которые группируются в динамические структуры.

Можно отметить и определенные недостатки ООП. Внедрение механизмов поддержки ООП усложняет языки программирования, иногда значительно. Ярким примером такого рода является язык Си++, который не ограничивается реализацией минимально необходимых механизмов наследования, а предусматривает обширный набор средств, некоторые из которых, как показала практика, избыточны и небезопасны.

Программа, написанная в объектно-ориентированном стиле, может оказаться менее понятной. Это обусловлено, в частности, использованием позднего связывания, когда по тексту программы невозможно определить, какая из процедур (функций, методов) будет вызвана.

Использование объектного подхода в значительной мере противоречит принципу нисходящей пошаговой детализации при разработке программ. Этот подход обеспечивает решение принципиальных вопросов общей организации программы на ранних стадиях проектирования, в то время как проектирование программы как совокупности объектов заставляет с самого начала больше внимания уделять деталям.

Наличие трудностей, привносимых в проектирование программ объектным подходом, подтверждается, например, таким обстоятельством. Со времени утверждения технологии структурного программирования практически перестали использоваться блок-схемы при разработке программ. Необходимость в них отпала, поскольку хорошо структурированная программа и так обладает достаточной выра-

зительностью и наглядностью. Ее текст непосредственным образом отражает порядок ее работы. При использовании же объектно-ориентированного проектирования вновь возникла потребность в дополнительном графическом языке, примером которого является известная «нотация Буча» [Буч, 1999], предусматривающая построение многочисленных диаграмм.

Язык программирования Си++

Си++ — один из самых известных и распространенных языков объектно-ориентированного программирования. Именно Си++ в конце 1980-х и начале 1990-х был в центре того ажиотажа, который возник вокруг ООП.

Язык формировался более 10 лет. В 1980 году сотрудник Bell Laboratories Бьерн Строуструп (Bjarne Stroustrup), стремясь улучшить Си, обеспечив поддержку абстракции данных и объектно-ориентированного программирования, ввел в Си понятие класса, заимствованное из языка Симула-67. Кроме того, были добавлены контроль и преобразование типов параметров функций и некоторые другие возможности. В течение нескольких лет язык, получивший название «Си с классами», использовался в исследовательской группе Б. Строуструпа.

В 1983–1984 годах в язык были внесены добавления: виртуальные функции, совместное использование (перегрузка, переопределение) операций (как в Алголе-68 и Аде). В новой реализации язык получил остроумное название Си++ (увеличенный, расширенный Си), предложенное Риком Маскитти (Rick Mascitti). В 1985 году Б. Строуструп опубликовал первую книгу с описанием Си++: «The C++ Programming Language», Addison-Wesley, 1985 (русский перевод: [Строуструп, 1991]). С этого времени началось распространение языка, стали появляться его реализации для разных платформ.

Интересно отметить, что в течение нескольких лет Си++ был реализован только с помощью конвертора в язык Си (транслятор cfront), и только в конце 1980-х появились компиляторы, не использующие Си в качестве промежуточного языка. Это объясняется, во-первых, желанием обеспечить простой перенос языка на разные платформы, используя мобильность Си — компиляторы Си существуют для любых систем; во-вторых, — сложностью Си++.

В дальнейшем язык постоянно расширялся. Были добавлены множественное наследование[24], шаблоны[25], обработка исключений. В 1989 году с созданием объединенного комитета ANSI/ISO начался процесс стандартизации языка, растянувшийся на долгие годы. В ходе стандартизации в язык вносились все новые добавления: логический тип, пространства имен и др. В 1993–94 годах в проект стандарта включено описание библиотеки ввода-вывода и библиотеки стандартных шаблонов (Standard Template Library, STL), предложенной сотрудником компании Hewlett Packard А. Степановым. Стандарт принят в 1998 году (ISO/IEC 14882-1998).

В 90-е годы было осуществлено большое число реализаций Си++. Системы программирования на языке Си++ были выпущены и активно продвигались компаниями Microsoft (Microsoft C++, Visual C++) и Borland (Turbo C++, Borland C++, C++ Builder). В компании Microsoft Си++ стал основным языком системного программирования. Получил признание компилятор Watcom C++, отличавшийся высокой степенью оптимизации создаваемого кода. Развитыми возможностями обладает IBM Visual Age C++. Широко известен компилятор g++, разработанный в рамках проекта GNU[26].

В конце 90-х обозначилась тенденция монополизации рынка компиляторов Си++. На платформе Windows доминирующее положение занял компилятор Visual C++ компании Microsoft. Располагая значительными ресурсами, Microsoft сумела довести продукт до высокой степени совершенства, обеспечив, в частности, высокий уровень оптимизации создаваемого компилятором машинного кода. Обладая форой перед конкурентами, Visual C++ обеспечивает и наилучшую интеграцию с системой Windows.

[24] При множественном наследовании производный класс может иметь более одного базового класса (непосредственного суперкласса). Последующая практика показала, что множественное наследование в Си++ создает больше проблем, чем преимуществ.

[25] Механизм шаблонов (templates) позволяет параметризовать типами функции и классы. Служит основой так называемого обобщенного программирования (generic programming). Подобен обобщенным (родовым) модулям языка Ада.

[26] Проект GNU (GNU's Not Unix — GNU не Unix), начатый в 1984 году, ставит целью создание свободного программного обеспечения, распространяемого в исходных текстах (www.gnu.org).

Высокую степень оптимизации кода, учет особенностей конкретных моделей микропроцессоров, выпускаемых компанией Intel, обеспечивает компилятор Intel C++.

Основные черты Си++

- Совместимость (не полная) с языком Си. Не каждая программа на Си является правильной программой на Си++. В некоторых отношениях (предварительное описание функций и др.) правила Си++ более строги. Многие системы программирования, однако, позволяют сочетать в одном проекте файлы, написанные на Си и Си++. В зависимости от расширения имени файла он компилируется в соответствии с правилами одного или другого языка.
- Отсутствует понятие модуля. Единицей компиляции является файл. Файлы компилируются независимо друг от друга. Взаимный контроль на этапе трансляции обеспечивается с помощью включения во входной поток транслятора так называемых заголовочных файлов. Многие соответствия между файлами могут быть проверены только на этапе компоновки (сборки) программы. Области действия имен могут дополнительно регулироваться конструкцией `namespace` (пространство имен).
- Основные типы данных, массивы и структуры, структурные управляющие операторы, правила записи и использования выражений аналогичны соответствующим элементам языка Си.
- Более строгий, чем в языке Си, статический (во время компиляции) контроль типов.
- Динамическая информация о типах (Run Time Type Information — RTTI) и контроль типов во время выполнения.
- Предусмотрена передача параметров функций по ссылке наряду с передачей по значению.
- Поддержка объектно-ориентированного программирования. Классы с гарантированной инициализацией объектов с помощью функций-конструкторов, множественное наследование (базовые и производные классы), динамический полиморфизм (виртуальные функции), абстрактные классы.
- Средства управления доступом к элементам классов: скрытые (`private`, приватные), защищенные (`protected`), общедоступные (`public`, публичные). Друзья (`friend`) классов — функции, не яв-

ляющиеся элементами классов, но имеющие доступ к их скрытым и защищенным элементам.

- Переопределение (совмещение, совместное использование) функций и операций. Одноименные функции с параметрами разного типа считаются различными. Встроенные операции, такие как «+», «–», могут быть определены для пользовательских типов.
- Встраиваемые (`inline`) функции. Компилятор помещает код встраиваемой функции в место ее вызова.
- Шаблоны классов и функций — средства параметризации классов и функций типами. Составляют основу так называемого обобщенного программирования. Подобны настраиваемым (родовым) модулям языка Ада.
- Средства обработки исключений.

Примеры программ на Си++

Начнем как обычно с простейшей законченной программы «Hello, World!». На Си++ она может быть неотличима от программы на классическом Си. Разве что придется перед текстом программы добавить директиву препроцессора `#include <stdio.h>`, которая присоединит к программе «заголовочный» файл `stdio.h`, содержащий предварительные описания (прототипы) стандартных функций ввода-вывода, в том числе и функции `printf`. В Си++ использовать функцию без предварительного описания нельзя. Такую директиву можно было поместить и в тексте на Си, но там она необязательна.

В стандартной библиотеке Си++ предусмотрены средства потокового ввода-вывода. С их помощью и запишем программу.

```
// Простейшая программа на Си++
#include <iostream.h>
main ()
{
   cout << "Hello, World!\n";
}
```

Во-первых, можно обратить внимание на новый вариант оформления комментария. Символы `//` начинают комментарий, который заканчивается в конце той же строки.

Директива препроцессора `#include <iostream.h>` подключает к программе заголовочный файл, содержащий описания базовых средств потокового ввода-вывода.

Единственный оператор программы передает строку "Hello, World!\n" в стандартный выходной поток, обозначаемый переменной cout. Эта переменная класса ostream (выходной поток) определена в iostream.h. Операция << означает запись данных в выходной поток. Содержимое строки, записанной справа, как бы передается в записанную слева переменную cout.

Однако не надо думать, что << — это предопределенная операция Си++. Операции ввода-вывода вообще не являются частью языка Си++. В Си++, как и в Си, операция << обозначает побитовый сдвиг влево. Но в заголовочном файле iostream.h и программах стандартной библиотеки, реализующих помещенные в iostream.h декларации, эта операция переопределена для случая, когда ее левый операнд относится к классу ostream. Переопределена средствами Си++, предусматривающими совместное использование операций (overloading).

Обобщенная сортировка на Си++

Вообще-то функция, выполняющая сортировку вещественного массива, записанная на Си++, оказывается очень похожей на соответствующую программу на Си. Но в Си++ есть средство, которое позволяет почти не усложняя программу сделать ее пригодной для сортировки массивов любого типа, не только вещественного. Это средство — шаблоны. Однажды написанная функция сортировки может применяться к массивам с элементами любого типа, допускающего присваивание и сравнение «на меньше». Если это пользовательский тип (класс), то программист может сам переопределить стандартные операции «=» и «<» для такого класса.

Демонстрируя применение шаблона, я использовал самый простой вариант сортировки вставками и самый понятный способ его записи. Хитроумные конструкции Си (они разрешены и в Си++), использованные в предыдущих примерах, здесь не применяются.

```
// Шаблонная функция сортировки вставками
template <typename T>
void InsSort(T a[], int n) {
   for (int i = 1; i<n; i++)
   { T x; int j;
      x = a[i]; j=i-1;
      while (j>=0 && x<a[j]) {
         a[j+1] = a[j];
```

```
            j--;
        }
        a[j+1] = x;
    }
 }
```

Запись template[27] <typename T> вместе с упоминанием T в списке параметров функции InsSort делает имя типа T как бы дополнительным параметром этой функции, но параметром необычным. Его фактический смысл определяется типом элементов того массива, который будет упорядочиваться. Например, если имеются такие массивы:

```
float a[1000]; int b[1000];
```

то при вызове

```
InsSort(a, 1000);
```

в роли T будет выступать тип float, а при вызове

```
InsSort(b, 1000);
```

вместо типа T будет принят тип int.

Текст шаблонной функции записывается лишь однажды. Но для каждого типа элементов массива, использованного при обращении к InsSort, компилятор создаст свой экземпляр машинного кода функции.

Оценка Си++

Си++ по-прежнему является одним из основных языков программирования. Однако его оценка с годами менялась. Если в начале 1990-х на волне объектно-ориентированной моды критика Си++ была почти не слышна, то, начиная с 1995 года, оценки языка программистским сообществом становятся более трезвыми. Этому способствовало появление языка Ява (Java), предложенного компанией Sun Microsystems. Сделав Яву внешне похожим на Си++, разработчики Sun предприняли попытку избавить новый язык от изъянов предшественника. С тех пор не замечать недостатки Си++ стало невозможным. Назову эти недостатки.

- Чрезмерная сложность. В ходе развития языка он постоянно расширялся. И автор языка, и участники процесса стандартизации не смогли удержаться от добавления все новых и новых средств, мно-

[27] Template — шаблон.

гие из которых являются избыточными или даже оцениваются как бессмысленные (например, ключевое слово mutable). Даже разработчики компиляторов Си++, которые, безусловно, являются экспертами по языку, заявляют: «ни один человек сейчас не в состоянии точно помнить все его детали и тонкости» [Зуев, 1996].

- Запутанный, плохо формализуемый синтаксис. Нетривиальная семантика многих конструкций. Это создает трудности, как при освоении, так и при реализации Си++. Чего только стоит фраза из проекта стандарта Си++ [Эллис, 1992]: «Если нечто выглядит как объявление, это оно и есть…» Си++ изобилует внешне привлекательными и несложными на первый взгляд конструкциями, точные правила трактовки которых оказываются в действительности запутаны и сложны. Неучет или неточное понимание программистом этих правил может приводить к труднообнаруживаемым ошибкам. К примеру, описание правил совместного использования (overloading, перегрузка, переопределение) функций и операций в упомянутом проекте стандарта занимает более 30 страниц. Это больше, чем, например, полная спецификация языка Оберон.
- Вынужденная совместимость с языком Си. Си++ создавался в развитие языка Си. Совместимость с Си, безусловно, стала одной из причин широкой популярности Си++. Но необходимость сохранения совместимости не позволила Си++ стать языком, обеспечивающим наряду с высоким уровнем абстракции достаточную надежность и безопасность. Си++ вынужденно унаследовал от Си заведомо небезопасные средства низкого уровня (адресная арифметика и др.). Совместимость с Си — один из основных источников ненадежности Си++. В Си++ сохраняются и все средства языка Си, провоцирующие создание хитроумно устроенных запутанных программ.
- Неполный контроль типов. Хотя при создании Си++ были предприняты шаги по усилению статического контроля соответствия типов, этот контроль не является исчерпывающим. По-прежнему неотличимы целый и символьный типы, массивы и ссылки. С помощью операции приведения разрешены почти любые взаимные преобразования. Введение в язык логического типа (`bool`) носит чисто декоративный характер, поскольку этот тип в Си++ неотличим от целого и не обеспечивает, в отличие от языков со строгой типизацией, где он играет важную роль, никакой дополнительной защиты от ошибок.

Названными недостатками Си++ может быть объяснена, по-видимому, невысокая стабильность работы некоторых широко распространенных коммерческих систем программного обеспечения, реализованных на этом языке.

Появление языка Ява в 1995 году и языка Си# («Си шарп») в 2000-м рассматривалось как признание несостоятельности Си++[28]. Оба эти языка, основанные на Си++, не только не расширяют его какими-либо принципиально новыми средствами, а, наоборот, во многом сужают, упрощают, делают более строгим.

Сказанное, однако, не мешает «неправильному» языку Си++ оставаться фактическим стандартом. Его стиль импонирует вкусам и привычкам широкого круга практических программистов. Компиляторы Си++ обеспечивают получение эффективного машинного кода. Большое количество программ, написанных на Си и Си++ и требующих развития и спровождения, сохраняют язык востребованным.

Язык программирования Оберон

История Оберона

Оберон является результатом эволюции линии языков, берущей начало от Алгола-60. Он создан профессором Швейцарского федерального технологического университета (ETH) Никлаусом Виртом, автором Паскаля и Модулы-2.

В 1985 году в ETH началась работа над проектом операционной системы для однопользовательской рабочей станции Ceres – компьютера, базирующегося на RISC-процессоре семейства NS32000. Ceres (по-русски — Церера) — имя древнеримской богини земледелия и плодородия и одновременно название первого по времени открытия (1801 г.) и самого крупного по размерам астероида. Выбрав для рабочей станции название малой планеты, авторы проекта несомненно хотели подчеркнуть необходимость ограничить сложность системы, сохранить ее обозримой.

Система была задумана как набор раздельно компилируемых модулей-компонентов с тщательно проработанными программными ин-

[28] В противовес этому утверждению можно было бы заметить, что появление Модулы-2 и Оберона означало признание несостоятельности Паскаля. Во многом так и есть. Но Модула и Оберон обогатили Паскаль концепцией модулей (АТД) и расширением типов (ООП), в то время как основной мотив Явы и Си# — упрощение и упорядочение Си++.

терфейсами. А разработка приложений в такой системе должна сводиться к расширению этого основного набора.

Первоначально планировалось использовать язык Модула-2. Но по мере продвижения проекта становилось все более очевидно, что кроме расширяемости системы в процедурном смысле необходима и поддержка расширяемости типов данных. В Модуле же не предусмотрено определение одного типа данных как расширения уже существующего. Эти потребности стимулировали дополнение языка средствами расширения типов.

С другой стороны, в новой системе было решено использовать для управления памятью механизм сбора мусора. В этом случае программист освобождается от нетривиальной и чреватой ошибками работы по правильному освобождению блоков динамической памяти. За него эту работу выполняет специальная подсистема – сборщик мусора, который вступает в работу, когда при очередном запросе блока динамической памяти ее не хватает. «Мусорщик» пытается обнаружить в памяти «брошенные» блоки, то есть такие, на которые нет ссылок из загруженных в данный момент модулей, и возвращает эти блоки в свободную для распределения «кучу». Все, что нужно сделать программисту, чтобы созданная им динамическая структура (сколь угодно сложно организованный список, дерево и т. п.) была утилизирована, так это присвоить указателю на эту структуру значение `NIL`. При необходимости сборщик мусора может вызываться из программы и явно.

При реализации механизма сбора мусора необходимо обеспечить мусорщика во время выполнения программы необходимой информацией, как о блоках распределенной памяти, так и о размещении в памяти указателей, которые на эти блоки могут ссылаться. Наличие в языке Модула-2 записей с вариантами затрудняло решение этой задачи. Средства расширения типов, которые должны были дополнить язык, делали записи с вариантами лишними, и от них решено было избавиться. Но как только вы ограничиваете язык, что-либо удаляя из него, теряется совместимость с прежними версиями. Старые программы не будут работать в новой системе. Появление нового языка стало неизбежным.

Оберон носит имя одного из спутников Урана[29]. К выбору такого названия Вирта подтолкнуло событие, произошедшее в то время, ко-

[29] Спутники Урана названы именами шекспировских героев. Оберон — царь эльфов и фей из комедии Шекспира «Сон в летнюю ночь».

гда он работал над проектом языка и операционной системы. В январе 1986 года американский космический аппарат «Вояджер-2», пролетая мимо Урана, сделал снимки планеты и ее спутников. Был сфотографирован и Оберон. На Вирта сильное впечатление произвела филигранная точность эксперимента, в ходе которого немалую роль сыграл бортовой компьютер «Вояджера». И это при том, что до сближения с Ураном аппарат находился в полете почти 10 лет. К тому же спутник Оберон был открыт англичанином Вильямом Гершелем в 1787 году, то есть за двести лет до описываемых событий.

Характеристика языка Оберон

Описание Оберона было опубликовано Виртом в 1988 году. Больше всего язык похож на своего непосредственного предшественника — Модулу-2. Н. Вирт подчеркивает, что Оберон получен изъятием из Модулы многого, и добавлением лишь некоторых усовершенствований. Из Модулы-2 удалены:

- Записи с вариантами.
- Непрозрачный (скрытый) экспорт типов.
- Перечислимые типы.
- Ограниченные типы (диапазоны).
- Множества общего вида. Оставлены только множества из ограниченного диапазона целых, которые являются лишь высокоуровневым средством манипулирования битами.
- Тип `CARDINAL`.
- Указатели не на записи и массивы.
- Массивы с нецелочисленными индексами и отличной от нуля нижней границей.
- Локальные модули.
- Не уточненный именем модуля импорт идентификаторов.
- Модули определений, главный модуль и понятие главной программы.
- Прежняя форма оператора **`WITH`**.
- Оператор **`FOR`**.
- Типы `ADDRESS` и `WORD` (заменены типом `BYTE`) и адресная арифметика; преобразование типов, обозначаемое идентификатором типа.
- Средства параллельного программирования.

Новые возможности, появившиеся в Обероне:

- Средства объектно-ориентированного программирования: расширение типов, проверка и охрана типа.
- Поглощение типов. Переход к тридцатидвухразрядным архитектурам определял большое разнообразие типов числовых данных, что сделало неудобным практически полный запрет Модулы-2 на присваивание неодинаковых типов. Числовые типы Оберона образуют иерархию: `SHORTINT <= INTEGER <= LONGINT <= REAL <= LONGREAL`. В этой цепочке значения «меньшего» типа могут быть присвоены переменным «большего» типа.
- Многомерные открытые массивы.
- Сборщик мусора.

На первый взгляд может показаться, что упрощения чрезмерны. Чего только стоит устранение цикла **`FOR`**. Но по некотором размышлении в этом можно увидеть пользу. Например, при освоении программирования одним из трудных моментов является приобретение навыков применения циклов с пред- и постусловием. А когда цикла **`FOR`**, а тем более **`GOTO`** нет, волей-неволей научишься применять **`WHILE`**, **`REPEAT`** и **`LOOP`**.

Аскетизм Оберона является его исключительно полезной чертой, особенно, если язык используется для обучения программированию. Сочетание простоты, строгости и неизбыточности предоставляет начинающему программисту великолепную возможность, не заблудившись в дебрях, выработать хороший стиль, освоив при этом и структурное, и объектно-ориентированное, и модульно-компонентное программирование.

Усовершенствована по сравнению с Модулой-2 структура программы. Вернее, понятия программы, как таковой, в Обероне вообще нет. Все, с чем вы имеете дело – это совокупность модулей-компонентов, которые загружаются в память динамически. Инициировать выполнение можно вызовом команды, в качестве которой рассматривается любая экспортированная процедура без параметров. Экспорт в Обероне оформляется исключительно изящно. Достаточно после имени экспортируемого объекта (процедуры, константы, переменной, типа) поставить звездочку (*). При этом программисту не нужно вручную выписывать спецификацию модуля, рискуя внести несоответствие между спецификацией и реализацией. Перечень экспортированного — интерфейс модуля — создается автоматически.

Объектно-ориентированным языком делают Оберон средства расширения типов. При конструировании объектной модели языка Вирт уделил первоочередное внимание не внесению в язык модной терминологии, а поддержке реальных программистских потребностей. Слова object в Обероне вообще нет; объект – это просто расширяемая запись. Предусмотрены адекватные механизмы, позволяющие оперировать динамическими объектами, используя легальные и безопасные средства. В Обероне такими средствами являются проверка и охрана типа. Указатели и параметры-переменные типа запись могут иметь как статический (определенный при описании), так и динамический тип (соответствующий типу того объекта, на который в данный момент фактически ссылается указатель). Проверка типа позволяет узнать динамический тип объекта, а охрана – обратиться к фактически имеющимся полям, имея твердую гарантию их существования.

Одна из главных черт Оберона – строгость. Строгий контроль соответствия типов, унаследованный от Паскаля и Модулы-2, строгие правила экспорта и импорта, строгий синтаксис обращения к полям записей. И в то же время Оберон допускает гибкость при обращении с данными. Это относится к уже обсуждавшемуся расширению типов, а также к простой и красивой концепции поглощения типов.

Важнейшее свойство языка — обозримость. Даже после первого знакомства с его описанием, объем которого составляет всего около 20 страниц, у опытного программиста создается ощущение неплохого знания языка. Оберон, в отличие от таких языков как ПЛ/1, Алгол-68, Си++, Объектный Паскаль, Ява или Си# нетрудно знать в совершенстве и целиком. Такое свойство языка невозможно переоценить.

Оберон-2

Вирт взял эпиграфом к описанию Оберона высказывание А. Эйнштейна: «Make it as simple as possible, but not simpler» («Делай как можно проще, но не проще чем нужно»). Почти сразу после появления языка возникли предложения по его «улучшению». И конечно же, все они сводились к расширениям языка. В 1991 году ученик Н. Вирта Ханспетер Мёссенбёк (Hanspeter Mössenböck) опубликовал в ETH сообщение о языке Оберон-2. Поводом к «усовершенствованиям» Оберона служило отсутствие в языке того, что принято называть виртуальными методами. В языке Вирта индивидуальное поведение объектов можно обеспечить, используя в записях поля процедурного

типа. Преимущество этого подхода – отказ от включения в язык *еще одного* механизма, в значительной степени дублирующего уже имеющиеся возможности, но усложняющего и язык, и компилятор. Но все же Вирт принял такое усовершенствование и поставил свое имя на описании расширенного Оберона, за которым закрепилось название Оберон-2. Эпиграфа Эйнштейна на этом документе уже нет. Авторами описания языка Оберон-2 являются Х. Мёссенбёк и Н. Вирт. Оберон-2 считается фактическим стандартом языка. В дальнейшем, если это не оговорено особо, я буду под названием Оберон понимать именно Оберон-2.

Основным нововведением Оберона-2 являются связанные с типом процедуры (аналог виртуальных методов в Объектном Паскале и виртуальных функций в Си++). Авторы Оберона-2 не стали вводить новых терминов, а обошлись уже имеющимися. Весьма остроумно выстроен синтаксис описаний связанных процедур. Не потребовалось никаких новых служебных слов, а формальный параметр-приемник, обозначающий экземпляр объекта внутри такой процедуры, описывается явно. Описания связанных процедур располагаются отдельно от описания типа запись, с которым они связаны. Связь же устанавливается по типу параметра-приемника.

Кроме связанных процедур в язык внесены еще некоторые усовершенствования. Предусмотрен экспорт только для чтения. Если после имени в описании переменной или поля записи поставить знак «–» вместо «*», то это имя экспортируется, но соответствующая переменная или поле не могут быть изменены вне экспортирующего их модуля.

Другие изменения — расширение применения открытых массивов, которые теперь могут использоваться не только как формальные параметры, но и в качестве базового типа указателей; расширение оператора **WITH** и, наконец, возвращение в язык оператора **FOR**.

Спецификация Оберона-2 приведена в приложении.

Дубовые требования

Летом 1993 года в английском городке Кройдоне в отеле «Дубовый» собралось около 30 разработчиков компиляторов и прикладных программистов, чтобы согласовать единые требования к реализациям Оберона-2. Среди участников встречи были и двое наших соотече-

ственников. Результатом совещания стал документ, получивший название «Дубовые требования» (Oakwood Guidelines).

Авторы «Дубовых требований» понимали, что стремление разработчиков трансляторов «улучшить» Оберон необходимо ввести в цивилизованное русло, чтобы обеспечить насколько возможно совместимость различных реализаций языка. При этом они ссылались на негативный опыт стандартизации Модулы-2, когда процесс развивался спонтанно.

Действительно, потребность договориться о трактовке некоторых положений описания языка существовала. Кроме того, средства ввода-вывода не являются частью собственно языка и соглашение о составе и спецификации стандартных библиотек позволило бы значительно улучшить совместимость разных реализаций. Попытка решить оба эти вопроса сделана в «Дубовых требованиях». Но что вызывает удивление, так это внимание, которое авторы «Требований» уделили расширениям языка. Признавая, что единственным официальным документом, определяющим Оберон-2, по-прежнему является описание, публикуемое ETH, создатели «Требований» оправдывали свое большое внимание к расширениям тем, что эти расширения не предлагаются, а определены на случай, если кто-то захочет язык расширить (комплексными числами, к примеру), то уж пусть расширяет как сказано. Противоречивая позиция. Подписи Н. Вирта под этим документом нет.

Оберон в мире

Первые реализации Оберона появились в ETH в конце восьмидесятых. После разработки оригинальной Оберон-системы для компьютера Ceres были созданы ее модификации для многих платформ: Amiga, DECStation, HP700, Linux, MacII, PowerMac, RS6000, SPARC, SiliconGraphics, Windows.

Одной из самых известных Оберон-систем стал Oberon/F, переименованный позже в BlackBox Component Builder. Разработка Oberon/F начата в 1992 году цюрихской компанией Oberon microsystems, Inc., тесно связанной с ETH. В отличие от систем ETH, которые обладают оригинальным и не очень привычным пользовательским интерфейсом и являются почти самостоятельными операционными системами, Oberon/F поддерживал традиционный для Windows и Macintosh интерфейс.

С создания конвертора в ANSI Си начинала свою работу с Обероном уже упоминавшаяся новосибирская фирма xTech (впоследствии XDS, Excelsior). В числе ее Оберон-продуктов под общим названием XDS конверторы в Си и Си++, компиляторы Оберона для нескольких платформ.

Автор этих строк также причастен к реализации Оберона. Летом 1998 году мною была опубликована в Интернете предварительная версия компилятора JOB, который стал первой в мире системой, транслирующей с языка Оберон-2 в байт-код виртуальной машины языка Ява (JVM). Такая схема позволила интегрировать Оберон с бурно развивающейся технологией Ява, в частности использовать из программ, написанных на Обероне, многочисленные библиотеки языка Ява. С помощью компилятора JOB можно создавать как обычные программы (консольные и оконные приложения), так и аплеты — небольшие программы, встраиваемые в веб-страницы.

К началу XX века было создано более двух десятков различных Оберон-систем. Активней всего использовался Оберон в европейских университетах. Интерес к языку Оберон проявили в NASA, компаниях Boeing, Bosch, Siemens, Alcatel, Motorola, DEC, Apple, Sun. Обратила внимание на Оберон корпорация Microsoft, высказав заинтересованность в реализации Оберона для многоязыковой платформы .NET (Dot NET).

В дальнейшем, однако, усилия энтузиастов Оберона сошли на нет. Небольшим, в основном академическим, коллективам было невозможно конкурировать с гигантами компьютерной отрасли, сделавшими ставку на Яву и Си#. Несмотря на свои достоинства, Оберон не получил широкого распространения. Но даже если вы никогда не будете программировать на Обероне, знакомство с этим языком безусловно полезно и важно. Оберон представляет собой квинтэссенцию технологий, воплощенных в универсальных языках программирования.

- Оберон — самый простой язык высокого уровня, обладающий средствами структурного, модульно-компонентного и объектно-ориентированного программирования.
  - Простая модульная структура со строго регламентированным экспортом-импортом, исчерпывающим межмодульным контролем и автоматическим формированием интерфейса.

    - Простая, полная и ясная модель объектно-ориентированного программирования.
    - Совершенный механизм управления динамической памятью на основе сборщика мусора.
- Оберон — язык надежного программирования, содержащий совершенные механизмы контроля.
- Оберон — идеальный язык для обучения программированию.

Примеры программ на языке Оберон

В Обероне нет понятия главной программы или главного модуля. По замыслу авторов языка должна быть обеспечена возможность вызова из среды Оберон-системы любой процедуры без параметров, экспортированной любым модулем. Такая процедура называется командой. Поэтому первый пример законченного модуля, позволяющего напечатать «Hello, World!», таков:

```
(* Простейшая программа на Обероне *)
MODULE Hello;
IMPORT Out;

PROCEDURE Run*;
BEGIN
   Out.String("Hello, World!");
   Out.Ln      (* Перевод строки *)
END Run;

END Hello.
```

Средства ввода-вывода не являются частью языка Оберон. Модуль `Hello` импортирует средства модуля `Out`, который входит в стандартный набор, рекомендованный «Дубовыми требованиями» и имеется в любой Оберон-системе. Знак «*» после названия процедуры `Run` (это не обязательное название, можно было выбрать и другое) говорит о том, что это имя экспортируется модулем `Hello`, то есть доступно из тех модулей, которые импортируют `Hello`, и из среды Оберон-системы. После компиляции модуля `Hello` и выполнения в Оберон-среде команды `Hello.Run` модуль `Hello` загружается, и процедура `Run` выполняется.

Обратите внимание, что при вызове процедур `String` и `Ln`, импортированных из `Out`, используются их уточненные имена вида `ИмяМодуля.ИмяПроцедуры`. В Обероне предусмотрен только такой

вариант импорта. Несколько удлиняя запись, он обеспечивает полную надежность, исключая любые коллизии имен.

Многие системы программирования на Обероне не поддерживают или не полностью поддерживают концепцию команд. Это связано, в частности, с тем, что в таких системах предусматривается создание программ, способных работать вне Оберон-системы в среде Windows, Unix, JVM или на вебстранице. В таком случае программа «Hello, World!» будет оформлена по-другому.

Компилятор XDS, транслирующий как с Оберона, так и с Модулы-2, предусматривает наличие главного модуля, который помечается директивой `<*+ MAIN *>`, форма которой соответствует стандарту ISO на язык Модула-2.

```
<*+ MAIN *>
(* Простейшая программа для XDS Oberon-2 *)
MODULE Hello;
IMPORT Out;
BEGIN
  Out.String("Hello, World!");
  Out.Ln
END Hello.
```

Модуль `Hello` в этой редакции содержит инициализирующую часть (начинается словом `BEGIN`), которая выполняется при загрузке модулей в память. В системе XDS выполнение программы начинается с инициализирующей части модуля, помеченного как главный.

Формат простейшей программы для компилятора JOB определяется тем, что нужно обеспечить совместимость с виртуальной машиной языка Ява (JVM), в среде которой выполняются программы, полученные с помощью JOB. По правилам языка Ява (и JVM) выполнение программы начинается с метода (процедуры) с именем `main`.

```
(* Простейшая программа для компилятора JOB *)
MODULE Hello;
IMPORT javalang, Out;
PROCEDURE main*(VAR args: ARRAY OF javalang.PString);
BEGIN
   Out.String("Hello, World!"); Out.Ln;
END main;
END Hello.
```

Процедура `main` должна иметь параметр — массив из Ява-строк. Это требование тоже соблюдается в приведенном примере. Модуль

javalang (один из модулей совместимости компилятора JOB со средой JVM) импортируется для того, чтобы указать при описании массива-параметра нужный тип для его элементов.

После компиляции этого примера с помощью JOB его можно выполнить, вызвав виртуальную машину:

```
C:\Lang&Trans\OBERON>java Hello
Hello, World!
```

Сортировка на Обероне

В отличие от предыдущих примеров, на которых отражается специфика разных систем программирования на Обероне, процедура сортировки не связана со средой выполнения и поэтому едина.

```
(* Сортировка вставками на Обероне-2 *)
PROCEDURE InsSort*(VAR a: ARRAY OF REAL);
VAR
   i, j : INTEGER;
   x    : REAL;
BEGIN
   FOR i := 1 TO SHORT(LEN(a))-1 DO
      x := a[i];
      IF x<a[i-1] THEN
         a[i] := a[i-1];
         j := i;
         WHILE (j>0) & (x<a[j-1]) DO
            DEC(j);
            a[j+1] := a[j];
         END;
         a[j] := x;
      END;
   END;
END InsSort;
```

Этот текст почти не отличается от процедуры, записанной на языке Модула-2. На таком простом примере разница проявляется лишь в деталях.

Имя процедуры сопровождается знаком «*», что означает, что эта процедура экспортируется, а значит, доступна из других модулей. Заголовок экспортированной процедуры будет включен в интерфейс модуля, формируемый автоматически специальным инструментом Оберон-системы — смотрителем.

Для определения размера открытого массива используется стандартная функция `LEN(a)`, значение которой равно числу элементов в массиве. Номер первого элемента всегда равен 0, а последнего — `LEN(a)-1`. Поскольку функция `LEN` имеет тип `LONGINT`, то для приведения ее значения к типу параметр цикла — переменной `i`, использована функция `SHORT`, которая преобразует значения «длинных» типов к «коротким»: `LONGINT` к `INTEGER`, `INTEGER` к `SHORTINT`, `LONGREAL` к `REAL`. Другим решением могло быть использование в роли параметра цикла переменной типа `LONGINT`. Тогда `SHORT` не потребовалась бы[30].

Аплет на Обероне

Компилятор JOB позволяет программировать на Обероне-2 аплеты, которые могут исполняться на веб-страницах браузерами, поддерживающими язык Ява. Не имея возможности обсуждать аплеты подробно, рассмотрим простейший пример. Наш аплет будет рисовать в отведенном ему поле черный квадрат. Пользы от такого рисунка никакой, разве что появляется повод вспомнить знаменитейшую картину Казимира Малевича, которая произвела фурор в начале XX века. Зато даже на этом примере видна основная структура аплета, и можно пройти и понять весь путь от его компиляции до просмотра в веб-браузере. Текст этого примера я взял из комплекта компилятора JOB. «Черный квадрат» — это первый написанный на Обероне-2 аплет, успешно откомпилированный в байт-код JVM.

```
(* First JOB compiled applet!
   "Black Square"
   Designed (1913) by Kasimir Malevich
   Programmed (2.03.98) by S.Sverdlov

   <applet code="BlackSquare_App" width=200 height=200>
   </applet>
*)
MODULE BlackSquare;
IMPORT app:=javaapplet, awt:=javaawt;
TYPE
   App* = RECORD(app.Applet) END;
```

[30] В большинстве реализаций Оберона тип INTEGER — это двухбайтовое, а LONGINT — четырехбайтовое целое. Поэтому, если речь идет о массивах с числом элементов не больше 32767, то подходит первый вариант, иначе — второй.

```
PROCEDURE (VAR a : App) paint*(g: awt.PGraphics);
BEGIN
   g.fillRect(30,30,140,140);
END paint;

END BlackSquare.
```

В этом небольшом примере использованы элементы объектной технологии. В аплете определен тип App, который является расширением типа Applet из стандартного модуля javaapplet. С типом App связана процедура paint. По соглашениям, принятым в Ява-системе, именно такая процедура (метод) аплета (экземпляра аплета, объекта типа BlackSquare.App), расположенного на веб-странице, вызывается браузером, когда возникает необходимость нарисовать поле, предоставленное аплету. Размер этого поля и другие параметры аплета определяется значениями, указанными в теге <applet>, который вписывается в HTML-текст веб-страницы. Запись такого тега для «Черного квадрата» приведена в комментарии, размещенном в начале примера.

Рис. 1.16. Аплет «Черный квадрат» в окне appletviewer

При отладке аплетов можно использовать специальный «просмотрщик аплетов» `appletviewer` (рис. 1.16) — программу, входящую в комплект разработчика на языке Ява (JDK — Java Development Kit). Запустив appletviewer и передав ему в качестве параметра имя файла, содержащего тег `<applet>`, например, имя файла `black.o` с текстом нашего примера, можно увидеть как исполняется аплет:

```
C:\LANDT\OBERON>appletviewer black.o
```

Язык программирования Ява

Язык программирования Ява (Java[31]) создан в недрах компании Sun Microsystems в рамках исследовательского проекта Green, открытого в 1990 году. Целью проекта, осуществлявшегося в обстановке строгой секретности, было создание среды программирования для устройств бытовой электроники. В 1991 году одним из участников проекта Джеймсом Гослингом (James Gosling) был разработан язык, названный им Oak (Дуб). Первая версия компилятора была разработана к осени 1992 года.

Одним из основных мотивов проекта Green было создание технологии программного обеспечения, не зависящего от платформы. Это обосновывалось необходимостью разработки программ для устройств, построенных на различных микропроцессорах. В основу проектного решения была положена схема, предусматривающая интерпретацию платформно-независимого П-кода (называемого авторами проекта байт-кодом). Полученный после компиляции байт-код исполняется с помощью интерпретатора — виртуальной машины. Эта схема аналогична той, что впервые использована при реализации языка Паскаль в начале 70-х. Для каждого типа процессора (целевой платформы) требуется своя реализация виртуальной машины.

После создания прототипа устройства, программируемого по новой технологии, были предприняты попытки заинтересовать промышленность (в частности индустрию цифрового кабельного телевидения) в подобного рода системах. Эти попытки окончились неудачей. Как выход из тупика, в 1994 году была поддержана инициатива одного из основателей Sun Билла Джоя (Bill Joy) переориентировать технологию на Интернет, переживавший в ту пору бурный рост.

---

[31] Вообще-то, как говорят, это слово произносится «Джава». Может быть. Но если в русском языке уже присутствуют остров Ява (Java) и (мечта моей юности) мотоцикл Ява (Jawa), то почему бы и языку не называться так же.

Для демонстрации возможностей новой технологии при создании и просмотре веб-страниц был разработан на языке Oak браузер WebRunner. Он позволял выполнять аплеты — небольшие программы, встроенные в веб-страницу и дающие возможность демонстрировать не только статичные тексты и рисунки, но и динамические объекты. В начале 1995 года язык и технология получили новое название Java[32], а браузер WebRunner был переименован в HotJava.

Новая технология была продемонстрирована руководству компании Netscape, чей браузер Netscape Navigator был в то время самым популярным средством для работы в Интернете. Технология получила одобрение, и виртуальная Ява-машина (Java Virtual Machine, JVM) была встроена в Netscape Navigator, обеспечив возможность исполнения аплетов.

В 1995 году Ява была выпущена. Далее последовала беспрецедентная рекламная кампания, пропагандировавшая достоинства новой технологии. Техническое решение рекламировалось теми же способами, что и потребительские товары.

Благодаря энергичной пропаганде, общей неудовлетворенности языками Си и Си++, большому интересу ко всему, связанному с Интернетом, желанием многих компаний и потребителей найти противовес решениям Microsoft, технология Ява получила широкую известность, поддержку, и начала быстро распространяться.

Росту популярности технологии Ява способствовала и ее открытость. Компания Sun сделала общедоступными спецификации языка, виртуальной машины и библиотек, образующих интерфейс прикладного программирования (Java API). Бесплатно распространяется комплект разработчика на языке Ява (JDK — Java Development Kit), включающий компилятор с Явы в байт-код (программа `javac`), интерпретатор байт-кода (виртуальную машину, программа `java`), отладчик, дизассемблер, библиотеку классов, другие инструменты, необходимые для создания программ.

Девизы языка Ява

Продвигая новую технологию, ее создатели акцентировали внимание на нескольких основных свойствах языка Ява, представляя его как про-

[32] Java — название излюбленного американцами сорта кофе, зерна которого готовятся по технологии, возникшей на острове Ява.

стой, объектно-ориентированный, распределенный, интерпретируемый, надежный, безопасный, не зависящий от архитектуры, переносимый, высокопроизводительный, многопоточный и динамический язык.

- **Простота**. Ява — это упрощенный и упорядоченный язык Си++. Освоение Явы программистами, знакомыми с Си и Си++ не составит труда. В язык не включены редко используемые, трудные для понимания, запутанные возможности Си++, такие как множественное наследование, переопределение операций, автоматическое приведение типов. Использование сборщика мусора упрощает программисту работу с памятью.
- **Объектная ориентированность**. Ява — «чистый» объектно-ориентированный язык. Кроме элементарных данных — чисел, символов, логических значений, все остальное в программе на Яве — динамические объекты. Любой определенный пользователем тип — это класс.
- **Сетевые возможности**. Ява-система сопровождается обширными библиотеками для работы с различными сетевыми протоколами.
- **Надежность**. Ява, в отличие от Си++ и тем более Си, — язык со строгим статическим (во время компиляции) контролем типов. Предусмотрен и динамический (во время выполнения) контроль типов и ошибочных ситуаций. В языке Ява отсутствует адресная арифметика (возможность выполнения арифметических действий с указателями) — один из источников ошибок в Си и Си++.
- **Безопасность**. Язык не предусматривает никаких средств для непосредственной работы с аппаратурой компьютера. Отсутствие адресной арифметики не позволяет программе обратиться к областям памяти, которые ей не принадлежат. Выполнение программы происходит под управлением виртуальной машины, на которую также возложены ряд функций по обеспечению безопасности. Все это затрудняет создание таких программ как вирусы, которые могут наносить ущерб пользователю.
- **Независимость от архитектуры**[33]. Язык Ява разработан для использования в сетях, которые состоят из компьютеров с процессорами и операционными системами различных архитектур. Чтобы

[33] Кроме понятия «архитектура» часто используют термин «платформа», понимая под платформой совокупность аппаратной и программной архитектуры, в первую очередь сочетание операционной системы и типа процессора. Говорят, например, о платформе Wintel — (ОС Windows + процессор Intel).

программа, написанная на Яве, могла выполняться на любом компьютере сети, она компилируется в независимый от архитектуры формат — байт-код. В таком формате программы циркулируют в сети. На каждом компьютере байт-код исполняется виртуальной машиной, которая уже специфична для каждой архитектуры.

- **Переносимость**. Спецификация языка Ява практически не оставляет возможностей для разночтения. Зафиксированы характеристики всех числовых типов данных с указанием конкретных диапазонов их значений и особенностей выполнения каждой операции. Важнейшим компонентом технологии является набор стандартных библиотек, определяющих единый, не зависящий от платформы, интерфейс прикладного программирования. Все это должно гарантировать одинаковое поведение Ява-программы на любой системе без необходимости внесения в программу изменений.
- **Интерпретируемость.** Байт-код исполняется виртуальной машиной без явного преобразования в машинные команды, отдельный этап компоновки отсутствует. Это делает разработку более оперативной. Файлы байт-кода (файлы классов) содержат достаточно информации, доступной во время выполнения, что позволяет выполнять необходимый контроль и облегчает отладку.
- **Высокая эффективность**. Хотя скорость интерпретации байт-кода бывает достаточна, в тех случаях, когда требуется большая производительность, возможна компиляция байт-кода «на лету» в машинный код конкретного процессора. Это позволяет достичь почти такой же скорости выполнения, как у программ, полученных компиляцией с языков Си и Си++.
- **Многопоточность**. Ява содержит средства параллельного программирования, обеспечивая адекватные возможности решения многих задач реального времени.
- **Динамичность**. Ява предусматривает динамическую компоновку классов во время выполнения программы. Это позволяет модернизировать части программы, без повторной компиляции всей программы. Предусматривается также возможность динамического контроля типов во время выполнения.

Опыт использования новой технологии и дальнейшее развитие событий показали, что не все заявленные свойства в полной мере присущи языку Ява, и по многим из названных пунктов имеются проблемы.

Простота. Ко времени появления Явы чрезмерная сложность Си++ уже вполне осознавалась сообществом программистов. В языке Ява многие сложные и запутанные механизмы Си++ были отвергнуты. Однако объективные данные, характеризующие сложность языков, показывают, что тезис о том, что Ява существенно проще Си++, не вполне оправдан. Результаты этих исследований будут приведены далее. Да и вообще, разве можно назвать простым язык, спецификация которого — это книга объемом более 700 страниц. Кроме того, многие из программирующих на Си++, в действительности используют его просто как «улучшенный» Си, не применяя объектных возможностей. В то время как писать программы на языке Ява, не освоив объектно-ориентированного подхода, затруднительно. А методология ООП отнюдь не проста.

Объектная ориентированность. Стремление разработчиков Явы к чистой объектной ориентации, своего рода объектный экстремизм, лишили язык ряда полезных свойств.

Программа на Яве состоит только из классов. Отсутствует понятие модуля. Из-за этого классы играют две разные роли. Во-первых, класс — это тип объектов, во-вторых — контейнер, содержащий описания статических данных, не имеющих никакого отношения к объектам данного класса. Такое объединение представляется противоестественным, трудным для понимания и объяснения и концептуально ущербным.

Все, кроме элементарных данных — объекты. В том числе строки. Это уже не просто массивы символов. Записать `s[i]`, чтобы получить *i*-й символ строки `s` нельзя. Нужно отправить объекту `s` сообщение: `s.charAt(i)`. Уровень абстракции при этом, конечно, повышается, но насколько это оправдано?

Все объекты существуют только в динамической форме. Память для них распределяется в куче. Неоправданным представляется отказ от простого и эффективного статического и автоматического (стекового) распределения памяти под массивы и записи (структуры, объекты), что является одной из причин снижения эффективности Ява-программ.

Часто говорят, что в Яве нет указателей. Правильнее было бы сказать, что нет арифметики указателей. А переменные-массивы и переменные-объекты представляют в Яве именно указатели на массивы и указатели на объекты. Поэтому, например, массив из массивов ока-

зывается не матрицей, а массивом из указателей на массивы. Особой гибкости это не добавляет, а неудобства и путаницу создает. Многомерные массивы в обычном понимании стали вообще невозможны.

Все это издержки «чистой» объектной ориентации.

Надежность. Переняв принцип строгой типизации от языков, происходящих от Паскаля, Ява действительно обеспечивает высокую надежность при обращении с данными. Но проявила себя ненадежность иного рода.

Первые реализации технологии Ява (компилятор, виртуальная машина, библиотеки) не были избавлены от ошибок. Система в целом оказалась довольно громоздкой. Виртуальная машина языка Ява выполняет непростые функции. Кроме собственно интерпретации байт-кода должна обеспечиваться динамическая загрузка классов (в том числе по сети), контроль безопасности и т. д. Все это делает реализацию виртуальной машины для различных платформ не слишком простой задачей. Библиотеки, составляющие интерфейс прикладного программирования (API — Application Programming Interface) и являющиеся важнейшим компонентом технологии, также весьма сложны. В их составе заметную долю составляют так называемые «родные методы» (native methods) — подпрограммы, существующие в виде машинного кода того процессора, на котором работает виртуальная машина. Такие методы программируются, как правило, на языке Си (виртуальная машина по заявлению ее разработчиков также была запрограммирована на ANSI Си) и должны отлаживаться отдельно для каждой реализации виртуальной машины. Встраивание виртуальной машины в браузеры также сопряжено с возможностью внесения ошибок.

На начальном этапе развития технологии (примерно до 2000 года) пришлось столкнуться с тем, что отладка программ на языке Ява, в частности аплетов, была сопряжена с преодолением ошибок и нестыковок. Методом проб приходилось находить решения, которые бы приемлемо работали в различных браузерах.

Независимость от архитектуры и переносимость. Безусловно, язык Ява машинно-независим, не привязан ни к какой конкретной платформе. Однако в первые годы распространения технологии получить программу, которая бы одинаково работала в любой среде, оказывалось не так просто. В дальнейшем по мере развития и усовершенствования Ява-систем ситуация улучшилась.

Интерпретируемость и высокая эффективность. Первоначально предполагалось, что программы на Яве будут выполняться с помощью интерпретатора. Но программа не исполняется непосредственно по ее исходному тексту. Присутствует этап компиляции с языка Ява в байт-код. Получается, что от интерпретации Яве достались недостатки (низкая скорость выполнения), а многие преимущества (отсутствие промежуточных файлов, отсутствие затрат времени на компиляцию, отсутствие необходимости в отдельном инструменте-компиляторе) оказались невостребованными. Многие программисты предпочитают при создании интернет-приложений работать с системами, которые выполняют интерпретацию программы прямо по ее исходному тексту. В этом случае отпадает необходимость освоения и использования при разработке сразу нескольких инструментов, выполнения нескольких этапов обработки программы, не нужно возиться с множеством создаваемых компилятором файлов[34]. В результате популярностью пользуются системы, основанные на более примитивных языках, таких как JavaScript и Перл.

Устройство языка Ява

Хотя многие черты Ява уже названы, имеет смысл рассмотреть, кроме рекламных лозунгов, особенности технического устройства языка.

- Независимость от платформы. Язык не содержит каких-либо свойств, зависящих от программно-аппаратной платформы. Спецификация языка не предусматривает также характеристик, определяемых реализацией. Например, правила выполнения арифметических операций и способ представления числовых типов однозначно определяются спецификацией языка. В то же время предусматривается возможность использования родных (native) методов, которые программируются на других языках (обычно Си или Си++) и обеспечивают взаимодействие с конкретной платформой.
- Единицей компиляции является файл, который может содержать один или несколько классов. Классы распределяются по пакетам (`package`). Понятие модуля отсутствует. Разрешен неуточненный именем пакета импорт. Пакет `java.lang` является, по сути, частью языка.

[34] При компиляции одного файла, содержащего части программы на языке Ява, образуются несколько файлов байт-кода, по одному на каждый определенный в исходном файле класс.

- Классы — основная единица программы и основа объектной модели языка. Предусмотрено, в отличие от Си++, только одиночное наследование. Во время выполнения классы загружаются динамически.
- Интерфейсы — особая разновидность абстрактных классов, предоставляющих некоторую разновидность множественного наследования. Класс может реализовывать несколько интерфейсов.
- Основные типы данных: логический (`boolean`), символьный (`char` — 16-разрядные символы в кодировке Unicode), 8-разрядный (`byte`), 16-разрядный (`short`), 32-разрядный (`int`) и 64-разрядный (`long`) целый, 32-разрядный (`float`) и 64-разрядный (`double`) вещественный.
- Структуры данных — одномерные массивы и объекты (записи, экземпляры классов). Строки символов — разновидность объектов. Массивы также рассматриваются как особый вид объектов.
- Строгий статический и динамический контроль соответствия типов (как в паскалеподобных языках).
- Динамическое использование памяти — основной механизм. Память под массивы и записи (объекты) распределяется динамически в куче.
- Запрет адресной арифметики (операций с указателями). Понятие указателя (ссылки) и адреса отсутствуют, хотя переменные-объекты и переменные-массивы существуют именно в форме указателей на собственно объекты и массивы.
- Сборка мусора. Утилизация освобождаемой объектами памяти выполняется автоматически сборщиком мусора.
- Управляющие операторы (`if`, `while`, `do`, `for`, `switch`) подобны соответствующим конструкциям Си и Си++.
- Методы класса (статические методы) — аналог обычных процедур и функций. Методы экземпляра (виртуальные методы) — связанные с типом объекта процедуры и функции. Предусмотрены совместное использование (перегрузка, overloading) и переопределение (overriding) методов. Перегрузка предполагает возможность использования методов с одинаковым названием, но разными типами параметров, а переопределение означает, что методы суперкласса могут быть заменены в классе-наследнике. Параметры методов передаются только по значению.

В языке Си параметры также передаются по значению, но существует возможность в качестве фактического параметра указать ад-

рес, а формального — ссылку. В Си++ есть возможность передачи параметров по ссылке. Отсутствие в Яве адресов и ссылок и передача параметров только по значению не позволяют запрограммировать метод, имеющий выходные или изменяемые параметры примитивных типов (`int`, `char` и др.). На Яве нельзя, к примеру, написать метод для обмена значениями двух целых, подобный `procedure Swap(var x, y: integer)`. Необходимость решения этого вопроса породила одну из самых неуклюжих конструкций языка Ява — классы-фантики (wrapper classes).

- Потоки (threads) — средства параллельного программирования.
- Средства обработки исключений.
- Запутанный многословный синтаксис. Многие конструкции, унаследованные из языка Си, не способствуют получению понятных программ: операторы-выражения, стимулирующие побочный эффект, условная операция, большое число уровней приоритета операций. Управление областями действия (областями видимости) объектов программы при отсутствии ясной модульной структуры приводит к необходимости использования многочисленных модификаторов доступа.

Причины появления в Яве некоторых громоздких конструкций плохо объяснимы. Сравните, например, описание экспортируемой вещественной константы в Обероне и Яве. На Обероне: `CONST Pi*=3.14159;` на Яве: `public static final float Pi=3.14159f;`

Примеры программ на Яве

Программу «Hello, World» позаимствуем из книги авторов языка [Гослинг, 1997], написанной ими по образу и подобию знаменитой книги Б. Кернигана и Д. Ритчи о языке Си.

```
// Простейшая программа на языке Ява
class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

Класс `HelloWorld` исполняет в этом примере роль программного модуля, но не типа данных, поскольку не содержит ни полей, ни виртуальных методов. Единственный его элемент — статический метод (метод класса) `main`. Этот метод не возвращает никакого значения

(о чем свидетельствует описатель `void`) и имеет один параметр `args` — массив объектов типа (класса) `String`. Символьные строки в языке Ява — это объекты стандартного класса `String`, который определен в пакете `java.lang`. Пакет `java.lang` по умолчанию импортируется любой программой. С метода `main`, описанного так, как в нашем примере, и начинается выполнение программы (приложения) на языке Ява. Массив из строк `args` содержит параметры командной строки, которые могут быть переданы программе при запуске. В нашем примере эти параметры не используются.

Единственным оператором программы `HelloWorld` является вызов метода `println`, который печатает сообщение и выполняет перевод строки. Это метод класса `PrintStream`, к которому принадлежит статическое поле `out` класса `System`, обозначающее стандартный выходной поток. Класс `System` импортируется из пакета `java.lang`. Непростая конструкция, не правда ли?

Поместив приведенный текст в файл `HelloWorld.java`, его можно откомпилировать в байт-код, например, с помощью компилятора `javac` из состава JDK:

```
>javac HelloWorld.java
```

Полученный в результате файл класса `HelloWorld.class`, содержащий байт-код класса `HelloWorld`, можно выполнить с помощью интерпретатора `java` (виртуальной машины):

```
>java HelloWorld
Hello, World!
```

Ява-аплет

Добавление на веб-страницы динамического содержания с помощью небольших программ-аплетов, исполняемых браузером, — одна из возможностей технологии, которая позволила пропагандировать Яву как язык для Интернета.

Для примера возьмем аплет «Черный квадрат», который уже рассматривался при обсуждении языка Оберон. При запуске этот аплет рисует в отведенном ему поле размером 200×200 точек черный квадрат размером 140×140 (см. рис. 1.16).

```
/* Ява-аплет "Черный квадрат"
   <applet code="BlackSquare.class" width=200
height=200>
   </applet>
```

```
*/
import java.awt.*;
import java.applet.*;
public class BlackSquare extends Applet {
   public void paint(Graphics g) {
      g.fillRect(30,30,140,140);
   }
}
```

Здесь мы тоже имеем дело с одним классом, представляющим законченную программу. Но программа-аплет имеет устройство, отличное от программы-приложения (например, программы `HelloWorld`).

Класс `BlackSquare` определяет тип объекта, который будет создан браузером. В браузер, поддерживающий технологию Ява, встроен механизм, позволяющий вызывать в нужные моменты методы объектов стандартного класса `Applet` (передавать объектам класса `Applet` сообщения). Определенный нами класс `BlackSquare` расширяет класс `Applet` (`extends Applet`), переопределяя метод `paint` этого класса. Когда возникает необходимость перерисовать отведенное аплету поле, браузер передает объекту-аплету сообщение «paint», которое и обрабатывается запрограммированным нами методом. Класс `Applet` импортируется из стандартного пакета `java.applet`. Из пакета `java.awt`, содержащего средства создания оконного пользовательского интерфейса, импортирован класс `Graphics`, использованный для описания параметра `g` метода `paint`. С помощью этого параметра браузер передает аплету графический контекст, обеспечивающий возможности графического вывода.

Компиляция аплета выполняется аналогично компиляции приложения (например, `HelloWorld`), а отладку можно выполнить так же, как для аплета, написанного на Обероне. После компиляции в байт-код становится безразлично, был ли аплет изначально написан на Яве или на Обероне.

Время показало, что Ява-аплеты не получили широкого распространения. Гораздо чаще для размещения на веб-страницах динамических элементов применяются другие технологии. Начиная с 2015 года, в некоторых интернет-браузерах поддержка Ява-аплетов была прекращена.

Сортировка на Яве

Использование встроенных в язык Ява механизмов наследования, в частности интерфейсов, позволяет написать достаточно изощренный и универсальный вариант сортировки, но использование такой техники будет рассмотрено позже. Здесь же приведен метод, реализующий простой вариант сортировки вещественного массива.

```
// Сортировка вставками на Яве
public static void InsSort(float a[]) {
   for (int i = 1; i<a.length; i++) {
      float x = a[i];
      if( x<a[i-1] ) {
         a[i] = a[i-1];
         int j = i-2;
         while( j>=0 && x<a[j] )
            a[j+1] = a[j--];
         a[j+1] = x;
      }
   }
}
```

Нетрудно заметить, что программа очень похожа на текст, написанный на Си или Си++. Обратим внимание на отличия. Во-первых, массив — это объект. Его длину можно получить обращением к полю `length`, индексы начинаются с 0. Описания переменных могут располагаться в любом месте, поэтому локальные переменные `x`, `i` и `j` объявлены в момент их первого использования. Обратите внимание на оператор `a[j+1] = a[j--];` Поскольку в Яве гарантируется вычисление операндов всех операций (в том числе и операции присваивания) строго слева направо, уменьшение значения `j` произойдет только после использования этого значения в индексных выражениях левой и правой частей присваивания. В Си и Си++, порядок вычисления операндов не определен, поэтому такой оператор был бы неоднозначен, и следовало бы написать `a[j+1] = a[j]; j--;`

Модернизация языка Ява

Спецификация первой версии языка Ява была опубликована в 1996 году. В течение последующих лет в язык вносились изменения. В 1997 году были добавлены вложенные классы. В последующих версиях появились средства обобщенного программирования (generics),

перечислимые типы, цикл `foreach` и др. Существенно изменился и расширился набор библиотек.

Распространение языка Ява

Вначале программировать на Яве можно было с помощью набора инструментов, входящих в комплект JDK компании Sun. Но вскоре появились альтернативные системы программирования: Symantec Café, Microsoft J++, Borland JBuilder. Компания Sun также выпустила ряд систем, которые включали интегрированную среду разработки: Forte for Java, Sun ONE Studio, Sun Java Studio. В 2000 году новосибирской компанией Excelsior был выпущен оптимизирующий компилятор JET для Windows, транслирующий программы на Яве в машинный код процессоров Intel. Позднее стали свободно доступными развитые интегрированные среды разработки на языке Ява, такие как Eclipse, NetBeens, IntelliJ IDEA, Android Studio.

Язык Ява был представлен в первую очередь как средство программирования для Интернета. При этом уже в первых версиях JDK — инструментального комплекта средств разработки, содержались библиотеки для создания не только аплетов, но и приложений с оконным пользовательским интерфейсом. Такие программы тоже имели возможность взаимодействовать с Интернетом.

Благодаря бесплатному распространению компанией Sun средств разработки на Яве, возможность поэкспериментировать с аплетами получили множество программистов. Самым распространенным сюжетом для аплетов стали простенькие игры, играть в которые можно прямо на веб-странице.

Были предприняты и попытки реализовать на Яве крупномасштабные программные комплексы. Главным мотивом при этом было стремление получить не зависящие от платформы системы. Так, известная канадская компания Corel взялась за создание на Яве набора офисных приложений: текстовый процессор, электронная таблица и т. д. Проект завершился неудачей. Corel Office for Java ни по производительности, ни по функциональным возможностям, ни по устойчивости работы не мог соперничать с конкурирующими продуктами. Неудачей закончился и проект так называемого «сетевого компьютера». Его замысел состоял в том, чтобы уменьшить стоимость пользовательского рабочего места за счет того, что все необходимые программы (написанные на Яве) загружаются в такой компьютер по сети.

Выпущенные компанией Sun компьютеры JavaStation, способные исполнять Ява-программы и работающие под управлением Java OS, не получили распространения.

Оказалась непростой ситуация с аплетами. Появилось сразу несколько альтернативных технологий, позволяющих помещать на страницы динамические элементы, в том числе обеспечивающие интерактивное взаимодействие с пользователем. В первую очередь в ряду таких технологий надо назвать скриптовый язык JavaScript[35]. Программа на таком языке прямо в исходном виде включается в текст веб-страницы. На JavaScript можно написать в том числе и несложные игры. Как средство оживления веб-страниц, JavaScript стал намного популярней Явы.

С начала 2000-х годов можно было наблюдать продвижение Явы в сферу корпоративных информационных систем, на роль языка серверных приложений и в качестве средства программирования мобильных устройств. Эта роль оказалась очень важной. На многих веб-сайтах стали использоваться Ява-скриптлеты и Ява-сервлеты. Уже в 2003 году многие массовые модели мобильных телефонов были оснащены виртуальной Ява-машиной (упрощенный вариант для мобильных устройств) и могли использовать написанные на Яве программы.

Улучшились характеристики эффективности Ява-систем. Использование продвинутых технологий компиляции в сочетании с ростом быстродействия компьютеров позволило снять проблему низкой производительности.

Ява обладает рядом неоспоримых достоинств. Он, например, гораздо лучше подходит на роль языка для обучения программированию, чем несостоятельные в этом отношении Си и Си++.

К 2010-м годам Ява стал одним из самых распространенных языков программирования. Он широко используется в разработке больших распределенных программных систем, отдельные части которых могут работать на разных программно-аппаратных платформах. Успеху языка Ява способствует и то, что он является основным средством создания программ для Android — самой распространенной операционной системы мобильных устройств.

[35] Несмотря на название, JavaScript не имеет прямого отношения к Яве. Внешне программа на JavaScript чем-то напоминает текст на Паскале, чем-то — на Си, в чем-то похожа и на программу на Яве.

Язык программирования Си#

В июне 2000 года стало известно о новом языке программирования, родившемся в недрах компании Microsoft. Язык этот стал частью новой технологии Microsoft, названной .NET (читается «Dot Net»). В рамках этой технологии предусмотрена единая среда выполнения программ (Common Language Runtime, CLR), написанных на разных языках программирования. Одним из таких языков, основным в этой среде, и является Си# (C#, читается «C sharp», «Си шарп»). Названием языка, конечно же, хотели подчеркнуть его родство с Си++, ведь # — это два пересекшихся плюса. Но больше всего новый язык похож на Яву. И нет сомнений, что одной из причин его появления стало стремление Microsoft ответить на вызов компании Sun.

Хотя официально авторы Си# не были названы, но на титульном листе одной из предварительных редакций справочника по языку обозначены Андерс Хейльсберг (Anders Hejlsberg) — создатель Turbo Pascal и Delphi, перешедший в 1996 году в Microsoft, и Скотт Вилтамут (Scott Wiltamuth).

Единая среда выполнения программ основана на использовании промежуточного языка IL (Intermediate Language — промежуточный язык)[36], исполняющего почти ту же роль, что и байт-код виртуальной машины языка Ява. Используемые в рамках технологии .NET компиляторы с различных языков транслируют программы в IL-код. Так же как и байт-код, IL-код представляет собой команды гипотетической стековой вычислительной машины. Но есть и отличия в устройстве и использовании IL.

Во-первых, в отличие от JVM, IL не привязан к одному языку программирования. В составе предварительных версий Microsoft.NET имелись компиляторы с языков Си++, Си#, Visual Basic. Независимые разработчики могут добавлять другие языки, создавая компиляторы с этих языков в IL-код.

Во-вторых, IL предназначен не для программной интерпретации, а для последующей его компиляции в машинный код. Это позволяет достичь существенно большего быстродействия программ. Содержащие IL-код файлы несут достаточно информации для работы оптимизирующего компилятора.

[36] Идея применения единого промежуточного языка для построения многоязыковой системы программирования не нова. Еще в 60-е годы такие системы на основе общего машинно-ориентированного языка АЛМО были созданы в СССР для многих типов машин [Богданов, 1976].

Основные черты Си#

«Си# — простой, современный, объектно-ориентированный язык с безопасной системой типов, происходящий от Си и Си++. Си# будет удобен и понятен для программистов, знающих Си и Си++. Си# сочетает продуктивность Visual Basic и мощность Си++.» Такими словами начиналось описание Си#. Мы же рассмотрим технические особенности языка.

- *Единицей компиляции* является файл (как в Си, Си++, Яве). Файл может содержать одно или несколько описаний типов: классов (`class`), интерфейсов (`interface`), структур (`struct`), перечислений (`enum`), типов-делегатов (`delegate`) с указанием (или без указания) об их распределении по пространствам имен.
- *Пространства имен* (`namespace`) регулируют видимость объектов программы (как в Си++). Пространства имен могут быть вложенными. Разрешено употребление объектов программы без явного указания пространства имен, которому этот объект принадлежит. Достаточно лишь общего упоминания об использовании этого пространства имен в директиве `using`. Предусмотрены псевдонимы для названий пространств имен в директиве `using` (как в Обероне).
- *Элементарные типы* данных: 8-разрядные (`sbyte`, `byte`), 16-разрядные (`short`, `ushort`), 32-разрядные (`int`, `uint`) и 64-разрядные (`long`, `ulong`) целые со знаком и без знака, вещественные одиночной (`float`) и двойной (`double`) точности, символы Unicode (`char`), логический тип (`bool`, не совместим с целыми), десятичный тип, обеспечивающий точность 28 значащих цифр (`decimal`).
- *Структурированные типы*: классы и интерфейсы (как в Яве), одномерные и многомерные (в отличие от Явы) массивы, строки (`string`), структуры — почти то же, что и классы, но размещаемые не в куче и без наследования, перечисления, несовместимые с целыми (как в Паскале).
- *Типы-делегаты*, или просто «делегаты» (подобны процедурным типам в Модуле-2 и Обероне, указателям на функции в Си и Си++).
- *Типы подразделяются* на ссылочные (классы, интерфейсы, массивы, делегаты) и типы-значения (элементарные типы, перечисления, структуры). Объекты ссылочных типов размещаются в динамической памяти (куче), а переменные ссылочных типов являются, по сути, указателями на эти объекты. В случае типов-значений переменные представляют собой не указатели, а сами значения. Неяв-

ные преобразования типов разрешены только для случаев, когда они не нарушают систему безопасности типов и не приводят к потере информации. Все типы, включая элементарные, совместимы с типом `object`, который является базовым классом всех прочих типов. Предусмотрено неявное преобразование типов-значений к типу `object`, называемое упаковкой (boxing), и явное обратное преобразование — распаковка (unboxing).

- Автоматическая *сборка мусора* (как в Обероне и Яве).
- *Обширный набор операций* с 14 уровнями приоритета. *Переопределение операций* (как в Алголе-68, Аде, Си++). С помощью операторов `checked` и `unchecked` можно управлять контролем переполнения при выполнении операций с целыми.
- *Методы* с параметрами значениями, параметрами-ссылками (`ref`) и выходными параметрами (`out`). Слова `ref` и `out` нужно записывать перед параметром не только в описании метода, но и при вызове. Наличие выходных параметров позволяет контролировать выполнение определяющих присваиваний. По правилам **языка** любая переменная должна гарантированно получить значение до того, как будет предпринята попытка ее использования.
- *Управляющие операторы*: `if`, `switch`, `while`, `do`, `for`, `break`, `continue` (как в Си, Си++ и Яве). Оператор `foreach`, выполняющий цикл для каждого элемента «коллекции», несколько разновидностей оператора перехода `goto`.
- *Обработка исключений*.
- *Свойства* — элементы классов (объектов), доступ к которым осуществляется так же, как и к полям (можно присвоить или получить значение), но реализуется неявно вызываемыми подпрограммами `get` и `set` (как в Объектном Паскале — входном языке системы Delphi).
- *Индексаторы* — элементы классов (объектов), позволяющие обращаться к объектам так же, как к массивам (указанием индекса в квадратных скобках). Реализуются неявно вызываемыми подпрограммами `get` и `set`[37]. Например, доступ (для чтения) к символам

[37] Стремление обобщенно оформить доступ к массиву, сделав его синтаксически неотличимым от обращения к функции, можно найти в языке Ада, где для записи индексов используются круглые скобки. В случае с индексаторами Си# — наоборот, обращение к функции или процедуре маскируется под обращение к массиву.

строки может выполняться как к элементам массива благодаря тому, что для стандартного класса `string` реализован индексатор.

- *События* — элементы классов (поля или свойства) процедурного типа (делегаты), к которым вне класса, где они определены, применимы только операции `+=` и `-=`, позволяющие добавить или удалить методы-обработчики событий для объектов данного класса.
- *Небезопасный (unsafe) код*, использующий указатели и адресную арифметику, локализуется в частях программы, помеченных модификатором `unsafe`.
- *Атрибуты* — аннотации, которые можно добавлять к любой именованной единице — классу, полю, методу и т. д.
- *Препроцессор*, предусматривающий, в отличие от Си и Си++, только средства условной компиляции.

Примеры программ на Си#

Как обычно, рассмотрим вначале простейшую законченную программу, процесс ее компиляции и выполнения. Разместим текст программы в файле `Hello.cs`:

```
/* Простейшая программа на языке Си# */
class Hello {
   static void Main() {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Для компиляции программы можно воспользоваться компилятором `csc`, который входит в состав Microsoft .NET Framework SDK — комплект разработчика для среды Microsoft .NET и запускается из командной строки:

```
>csc Hello.cs
```

После компиляции будет получен исполняемый файл `Hello.exe`. Запустить его на компьютере, работающем под управлением ОС Windows, если на этом компьютере установлена поддержка Microsoft .NET. Дело в том, что полученный после компиляции файл (несмотря на свое название) содержит не обычные машинные команды, а IL-код, который будет преобразован в код процессора при загрузке и запуске программы. Запустив `Hello.exe`, получим:

```
>Hello.exe
Hello, World!
```

Теперь обратимся к тексту программы. В ней определен единственный класс `Hello`, в котором содержится описание статического метода (метода класса) `Main`. Статический метод с названием `Main` (прописные и строчные буквы в Си# различаются) является точкой входа в программу, написанную на Си#. С выполнения этого метода начинается работа программы. В отличие от языка Ява, метод `Main` в Си# может не иметь параметров, не важно также, возвращает ли он значение (являясь функцией) или нет. В нашем примере `Main` не имеет ни параметров, ни возвращаемого значения (`void`).

Единственный оператор в методе `Main` — вызов статического метода `WriteLine`. Это метод класса `Console`, предоставляющего доступ к стандартным выходному и входному потокам. Класс `Console` принадлежит (предопределенному) пространству имен `System`.

Для ссылки на класс `Console` использовано его полное название `System.Console` (уточненный идентификатор), включающее обозначение пространства имен `System`. Используя директиву `using`, можно сокращать запись, применяя не уточненные названием пространства имен обозначения:

```
/* Простейшая программа на языке Си# */
using System;   // разрешается неуточненный доступ
class Hello {
   static void Main() {
      Console.WriteLine("Hello, World!");
   }
}
```

Сортировка на Си#

Обсуждение сортировки простыми вставками не позволяет увидеть существенной разницы между Си# и языками-предшественниками — Си++ и Явой. Приведенный ниже листинг можно даже воспринимать как загадку. Найдите два отличия в этом тексте от соответствующего примера на языке Ява.

```
// Сортировка простыми вставками на Си#
public static void InsSort( float[] a ) {
   for( int i = 1; i<a.Length; i++ ) {
      float x = a[i];
      if( x<a[i-1] ) {
         a[i] = a[i-1];
         int j = i-2;
         while( j>=0 && x<a[j] )
```

```
                a[j+1] = a[j--];
            a[j+1] = x;
        }
    }
}
```

Отличия такие. Во-первых, при объявлении массива в Си# всегда надо писать `float[] a` (так же можно и на Яве, но там допустимо и `float a[]`). Во-вторых, слово «длина» пишется с большой буквы: `Length`. `Length` — это свойство (property) стандартного класса `System.Array`, который является родоначальником массивов в Си#.

Примеры использования объектной технологии

Перейдем теперь от разговора об идеях и языках объектно-ориентированного программирования к конкретным примерам.

Назначение встроенных в современные языки механизмов ООП сводится к тому, чтобы обеспечить построение динамических структур данных, состоящих из разнотипных элементов, обладающих индивидуальным поведением. Такие структуры называют гетерогенными[38]. Правильно выстроенные механизмы ООП должны обеспечивать безопасное манипулирование гетерогенными структурами и их элементами.

В качестве примера гетерогенной структуры возьмем очередь, в которую можно помещать элементы разных типов. Рассмотрим, как можно запрограммировать работу с гетерогенной очередью на разных языках: Обероне, Обероне-2, Яве и Си#.

Гетерогенная очередь на Обероне

Очередь — это структура данных, для которой при добавлении и извлечении элементов действует правило «первым пришел, первым ушел». По-другому это называют дисциплиной FIFO (First In First Out).

Удобно реализовать очередь с помощью списка. При этом естественно стремиться к такой реализации, которую не придется менять при изменении типа элементов, из которых очередь состоит. В нашем случае только о таком варианте и следует говорить, поскольку мы хо-

[38] Гетерогенный — неоднородный, состоящий из различных по своему составу частей.

тим, чтобы в одну и ту же очередь можно было помещать разнотипные данные.

Чтобы добавлять в конец очереди (списка), а извлекать из начала, предусмотрим указатели на первый (First) и последний (Last) элементы. Каждый элемент очереди будет содержать указатель на следующий (Next) элемент, обеспечивая тем самым связывание в единый список (рис. 1.17).

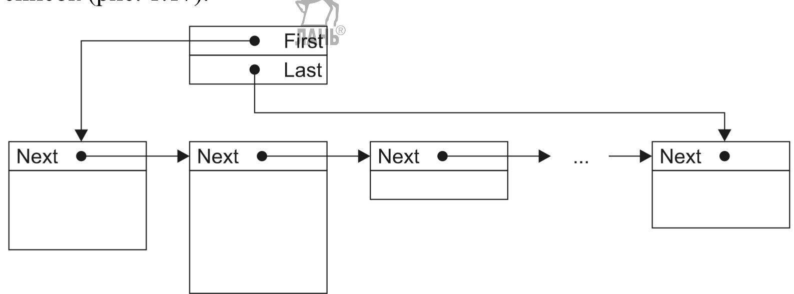


Рис. 1.17 Гетерогенная очередь

Обратите внимание на то, что элементы очереди изображены прямоугольниками разной величины, поскольку элементы разнотипны. Общим у всех элементов является лишь поле Next. Важно подчеркнуть, что в момент, когда мы собрались программировать операции с очередью, мы ничего не знаем (и не хотим знать) о конкретных типах элементов. Очередь должна быть универсальна.

Реализация очереди

Поместим описание (и реализацию) абстрактного[39] типа «очередь» в модуль `Queue` – «очередь».

```
MODULE Queue;
TYPE
   tNode* = POINTER TO tNodeDesc;
   tNodeDesc* = RECORD
      Next : tNode;
   END;
   tQueue* = RECORD
```

[39] Здесь «абстрактный» следует понимать, как «реализующий АТД», а не как абстрактный тип (класс) языков Ява или Си#.

```
      First, Last : tNode
   END;
```

Тип tNode — это указатель на элемент очереди (node — вершина, узел), а сам элемент имеет тип tNodeDesc — дескриптор (описатель) вершины. Звездочки, сопровождающие идентификаторы tNode, tNodeDesc и tQueue, означают, что эти типы экспортируются модулем Queue, то есть доступны для использования другими модулями. Принципиально важно, что имена полей Next, First и Last не экспортируются. Модули, использующие Queue, не будут (и не должны) знать деталей реализации очереди — соблюдаются основополагающие принципы АТД — отделение спецификации от реализации и упрятывание реализации. С другой стороны, наш модуль Queue не знает о типах элементов, которые фактически будут помещаться в очередь.

Продолжим реализацию (и спецификацию) очереди в модуле Queue. Предусмотрим процедуры для инициализации очереди (Init), добавления (Put) и извлечения (Get) элемента, а также логическую процедуру-функцию, позволяющую узнать, не пуста ли очередь (NotEmpty).

```
(* Инициализация *)
PROCEDURE Init*(VAR Q: tQueue);
BEGIN
   Q.First := NIL; Q.Last := NIL;
END Init;

(* Добавить в очередь *)
PROCEDURE Put*(VAR Q: tQueue; node: tNode);
BEGIN
   IF Q.First = NIL THEN
      Q.First := node;
   ELSE
      Q.Last.Next := node;
   END;
   Q.Last := node;
   node.Next := NIL;
END Put;

(* Удалить из очереди *)
PROCEDURE Get*(VAR Q: tQueue; VAR node: tNode);
BEGIN
   node := Q.First;
```

```
   Q.First := Q.First.Next;
   IF Q.First = NIL THEN Q.Last := NIL END;
END Get;

(* Очередь не пуста *)
PROCEDURE NotEmpty*(Q: tQueue): BOOLEAN;
BEGIN
   RETURN Q.First # NIL;
END NotEmpty;

END Queue.
```

Параметром процедур `Put` и `Get` является, наряду с очередью `Q`, указатель на добавляемую или извлекаемую вершину, названный `node`. Программа, использующая процедуру `Put`, перед добавлением в очередь элемента должна создать динамическую переменную, а указатель на эту переменную передать в качестве параметра процедуре `Put`. Предусмотреть создание динамической переменной-элемента списка внутри процедуры `Put` нельзя, поскольку тип добавляемого элемента процедуре `Put` неизвестен.

Вызов процедуры `Get` в случае, когда очередь пуста, недопустим и приведет к аварийному завершению программы. Перед обращением к `Get` следует предусмотреть (с помощью `NotEmpty`) проверку наличия в очереди элементов.

Программный интерфейс (перечень экспортируемых ресурсов) написанного нами модуля может быть получен автоматически с помощью специального инструмента Оберон-системы — смотрителя, либо формируется компилятором в ходе трансляции:

```
DEFINITION Queue;
TYPE
   tNode = POINTER TO tNodeDesc;
   tNodeDesc = RECORD END;
   tQueue = RECORD END;
PROCEDURE Init(VAR Q: tQueue);
PROCEDURE Put(VAR Q: tQueue; node: tNode);
PROCEDURE Get(VAR Q: tQueue; VAR node: tNode);
PROCEDURE NotEmpty(Q: tQueue): BOOLEAN;
END Queue.
```

Обратите внимание, что поля записей `tNodeDesc` и `tQueue` скрыты.

Использование очереди. Наследование

Теперь рассмотрим, каким образом можно использовать реализованную нами очередь. Будем записывать в очередь целые и вещественные числа. Для этого в программе, которая оперирует очередью (пусть это будет модуль Test) нужно определить два новых типа записей и два соответствующих типа указателей. Одна пара типов (запись и указатель на нее) будет представлять элемент очереди, содержащий целочисленное поле, другая пара — вещественное. Оба типа записей должны быть расширением tNodeDesc, а типы указателей — расширением типа tNode. Только в этом случае мы сможем обеспечить при вызове процедур Get и Put совместимость типов для параметра, обозначающего указатель на добавляемую или извлекаемую вершину (параметр node). По правилам языка Оберон, указатель типа tNode может ссылаться на переменную-запись, тип которой или tNodeDesc, или является расширением типа tNodeDesc.

```
MODULE Test;
IMPORT Queue, Out;
TYPE
   tInt = POINTER TO tIntDesc;
   tIntDesc = RECORD (Queue.tNodeDesc)
      x : INTEGER; (* Целочисленное поле *)
   END;
   tReal = POINTER TO tRealDesc;
   tRealDesc = RECORD (Queue.tNodeDesc)
      x : REAL;     (* Вещественное поле  *)
   END;
```

Модуль Test импортирует модуль Queue и модуль Out, содержащий элементарные средства вывода, которые потребуются нам для иллюстрации действий с очередью. Запись в скобках после слова **RECORD** означает, что определяемый тип является расширением типа, указанного в скобках (в нашем случае Queue.tNodeDesc — типа tNodeDesc из модуля Queue). Если tIntDesc и tRealDesc — расширения tNodeDesc, то соответствующие типы-указатели tInt и tReal считаются расширениями (наследниками) tNode. Записи tIntDesc и tRealDesc наследуют от типа tNodeDesc поле Next, хотя и «не знают» об этом. Запись tIntDesc расширяет тип tNodeDesc полем x целого типа, а tRealDesc — полем вещественного типа, также названным x.

Действия с очередью разместим в процедуре Run, которая является частью модуля Test. Предусмотрим описание необходимых переменных.

```
PROCEDURE Run*; (* Экспортируется *)
VAR
   Q    : Queue.tQueue; (* Очередь                   *)
   int  : tInt;         (* Указатель на цел. вершину *)
   real : tReal;        (* Указатель на вещ. вершину *)
   node : Queue.tNode;  (* Указатель на вершину      *)
   i    : INTEGER;      (* Счетчик цикла             *)
```

Заполнение очереди

Для иллюстрации будем помещать в очередь попеременно целые и вещественные значения. Запишем 10 пар. В качестве целого будем брать значение счетчика цикла i, в качестве вещественного — i/10.

```
Queue.Init(Q); (* Инициализация очереди Q *)
 (* Заполнение очереди *)
i := 1;
WHILE i<=10 DO
   NEW(int); int.x := i; Queue.Put(Q, int);
   NEW(real); real.x := i/10; Queue.Put(Q, real);
   INC(i);
END;
```

Добавление каждого элемента в очередь выполняется в три приема. Вначале с помощью вызова стандартной процедуры NEW создается динамическая переменная. При вызове NEW(int) она будет иметь тип tIntDesc, при вызове NEW(real) создается переменная типа tRealDesc. Затем полю x (в одном случае это поле целого типа, в другом — вещественного) присваивается значение. Наконец, вызовом процедуры Put элемент добавляется в очередь. Следует обратить внимание, что при двух вызовах Put вместо параметра node этой процедуры подставляются указатели разных типов (tInt и tReal). Но оба эти типа — расширения tNode, поэтому такая подстановка допустима.

Обслуживание очереди. Проверка и охрана типа

Теперь предусмотрим (в процедуре Run) обслуживание очереди, которое будет состоять просто в печати значений содержащихся в ней чисел (с удалением элементов из очереди). При этом мы, конечно, не вправе пользоваться знаниями о том, в каком порядке в очере-

ди следуют целые и вещественные. Однако будем считать, что кроме целых и вещественных других типов значений в очереди нет.

```
WHILE Queue.NotEmpty(Q) DO
   Queue.Get( Q, node );
   (* Напечатать значение элемента *)
      …
   Out.Ln; (* Переход на новую строку *)
END;
```

Это лишь набросок цикла обслуживания очереди. Вместо фрагмента, где должна быть печать извлеченного из очереди значения — многоточие. Рассмотрим, как следует заполнить этот пробел. Во-первых, обратите внимание, что при обращении к процедуре `Get` вместо выходного параметра подставлена переменная-указатель `node` типа `tNode`. Это единственно возможное решение в этой ситуации. Записать `Queue.Get(Q, int)` или `Queue.Get(Q, real)` нельзя по той простой причине, что мы не знаем, какого типа будет очередной элемент. Кроме того, это нарушило бы правило совместимости типов Оберона и привело бы к ошибке при компиляции.

Чтобы напечатать значение числового поля полученного из очереди элемента придется проверить тип этого элемента, точнее — тип соответствующего указателя. Для этой цели используем операцию проверки типа языка Оберон — **IS**.

```
IF node IS tInt THEN (*Печатать целочисленное поле x*)
   …
ELSE (* Печатать вещественное поле x *)
   …
END;
```

Условие `node IS tInt` истинно, если указатель `node` ссылается на элемент типа `tIntDesc`, то есть `node` имеет динамический тип `tInt`.

Как же выполнить печать, когда тип полученного элемента уже выяснен? Во-первых, модуль `Out` предоставляет две разных процедуры для печати целых и вещественных — `Out.Int` и `Out.Real` соответственно. `Out.Int(node.x, 16)` (здесь 16 — ширина поля вывода) нельзя. Дело в том, что переменная `node` имеет статический тип `tNode`, который не предусматривает поля `x` у той записи, указателем на которую он является. Поэтому обращение `node.x` некорректно. Хотя мы знаем (благодаря предусмотренной проверке), что фактически `node` будет ссылаться в первом случае на запись с целочислен-

ным, а во втором — с вещественным полем x. Воспользоваться этим можно, лишь предусмотрев контроль фактического типа node непосредственно в момент использования node для обращения к полю x. Конструкция Оберона, обеспечивающая такой контроль, называется «охрана типа». Используя охрану, получим:

```
IF node IS tInt THEN
   Out.Int( node(tInt).x, 16)
ELSE
   Out.Real( node(tReal).x, 16)
END;
```

Запись node(tInt) означает, что при выполнении программы будет проверен фактический тип node. Если этот тип — tInt (или расширение tInt), то переменная node рассматривается как имеющая тип tInt (то есть обозначение node(tInt) представляет переменную типа tInt) и, следовательно, наличие целочисленного поля x у записи, на которую ссылается node, в этом случае гарантировано. В этом и состоит смысл охраны типа — она обеспечивает безопасное обращение с указателями в строгом соответствии с их фактическим типом. Если динамический тип node не равен tInt и не является расширением tInt, при выполнении программы происходит ошибка.

Целиком фрагмент, отвечающий за обслуживание очереди, теперь выглядит так:

```
WHILE Queue.NotEmpty(Q) DO
   Queue.Get(Q, node);
   IF node IS tInt THEN
      Out.Int(node(tInt).x, 16)
   ELSE
      Out.Real(node(tReal).x, 16)
   END;
   Out.Ln;
END;
```

Такое решение, однако, может вызывать критику, поскольку не соответствует принципам объектного подхода. Выяснение типа объекта, полученного из очереди, и выбор варианта поведения происходят на верхнем уровне управления. Тем самым объекты лишаются активной роли. Вместо этого следовало бы просто передать объекту сообщение: «А ну-ка, напечатайся». Организацией такого решения мы сейчас и займемся.

Обработка сообщений на Обероне

Общая схема обслуживания очереди, основанная на передаче сообщений с переносом активности на уровень объектов, будет выглядеть так:

```
WHILE Queue.NotEmpty(Q) DO
   Queue.Get(Q, node);
   (* Передать объекту, на который ссылается
     указатель node, сообщение "Напечатать значение" *)
      …
   Out.Ln;
END;
```

Передача сообщения будет выполняться с помощью вызова процедуры (которая, наверняка, будет называться `Print` — печатать). Эта процедура могла бы выглядеть так:

```
PROCEDURE Print(node: Queue.tNode);
BEGIN
   IF node IS tInt THEN
      Out.Int(node(tInt).x, 16)
   ELSE
      Out.Real(node(tReal).x, 16)
   END
END Print;
```

Но в этом случае объекты снова лишаются самостоятельности. Просто появляется промежуточный уровень в виде процедуры `Print`, на который и переносится решение всех вопросов, связанных с печатью значений. Плохо и то, что при появлении новых типов объектов, которые мы захотим помещать в очередь, и значения которых будем печатать, придется переписывать (и перекомпилировать) процедуру `Print`. То есть добавление нового программного кода (новых типов объектов) будет вынуждать нас менять уже существующий код. Лучше бы без этого обходиться, поскольку такая ситуация способствует снижению надежности программы из-за необходимости постоянной модернизации уже написанного и отлаженного кода. Возможность расширять программу, не вторгаясь в уже существующие ее части, крайне важна[40]. И рассматриваемая нами технология это позволяет.

[40] Я бы назвал это принципом «аддитивного программирования»: добавляя, не меняй старого.

Предусмотрим у каждого объекта, который мы добавляем в очередь, поле процедурного типа, значением которого будет процедура (своя для каждого типа объекта), способная напечатать значение объекта данного типа.

Для объекта, содержащего целочисленное значение (объекта типа `tInt`[41]), это будет следующая процедура:

```
PROCEDURE  PrintInt(node: Queue.tNode);
BEGIN
   Out.Int(node(tInt).x, 16);
END PrintInt;
```

Может показаться, что параметр `node` этой процедуры можно было сделать типа `tInt`, и тогда не потребовалось бы использовать охрану типа при обращении к полю `x`. Ведь охрана типа приводит к дополнительным затратам во время выполнения программы. Но делать так не следует, поскольку все процедуры, исполняющие роль обработчиков сообщений «напечатать», должны иметь один и тот же тип параметра, то есть быть одинакового процедурного типа.

Для объектов типа `tReal` роль обработчика сообщений будет выполнять процедура `PrintReal`.

```
PROCEDURE  PrintReal(node: Queue.tNode);
BEGIN
   Out.Real(node(tReal).x, 16);
END PrintReal;
```

Если считать, что каждый объект (объект каждого типа), который мы записываем в очередь и извлекаем оттуда, должен уметь напечатать свое значение, то поле процедурного типа (это будет поле `Print`), означающее процедуру, которая печатает значение объекта, следовало бы поместить в описание типа `tNodeDesc` в модуле `Queue`. Но делать так мы не будем потому, что это противоречило бы первоначальному замыслу: создать и никогда больше не изменять модуль для работы с очередью. Ведь кроме необходимости печатать, может возникнуть и сколько угодно других потребностей при работе с объектами. И каждый раз менять модуль `Queue`? Ни в коем случае!

---

[41] Точнее было бы говорить об объекте типа `tIntDesc` — таков тип соответствующей записи. `tInt` — это тип указателя на объект. Но поскольку в нашем примере для обращения к объектам используются исключительно указатели, будем говорить для простоты об объектах типа `tInt`, `tReal` и т. п.

С другой стороны, неразумно помещать поле `Print` и в описание каждого конкретного типа объектов (`tIntDesc`, `tRealDesc`). Это поле общее для таких объектов, общее и по названию и по типу, поэтому было бы правильно, если б объекты этих типов унаследовали поле `Print` от общего типа-предка. Кроме того, если поля `Print`, пусть и одноименные (как поля `x`), определить для каждого типа в отдельности, то для обращения к этим полям снова пришлось бы различать типы объектов прямо в цикле обслуживания очереди, от чего мы стремимся уйти.

Назовем новый, общий для `tInt` и `tReal`, тип-предок `tPrintable` — тот, который «умеет» напечатать свое значение. Его описание поместим в разделе типов модуля `Test`, который теперь будет выглядеть так:

```
TYPE
   tPrintable = POINTER TO tPrintDesc;
   tPrintDesc = RECORD (Queue.tNodeDesc);
      Print : PROCEDURE (node: Queue.tNode);
   END;

   tReal = POINTER TO tRealDesc;
   tRealDesc = RECORD (tPrintDesc)
      x : REAL;
   END;

   tInt = POINTER TO tIntDesc;
   tIntDesc = RECORD (tPrintDesc)
      x : INTEGER;
   END;
```

Типы `tInt` и `tReal` стали (непосредственными) расширениями `tPrintable`, а уже этот тип — непосредственное расширение типа `tNode` из модуля `Queue`. `tInt` и `tReal` остаются при этом (не непосредственными) расширениями `tNode` (рис. 1.18).

Замечу, что программа никогда не будет создавать объекты типа `tPrintable`. В записи соответствующего типа нет даже поля, содержащего значение объекта (вроде поля `x`). Отсутствует и процедура (вроде `PrintInt` или `PrintReal`), которая способна печатать значение объектов такого типа. Подобные типы, которые служат лишь промежуточными звеньями в иерархии наследования, являясь носителем некоторых общих для своих наследников свойств (спецификации свойств), часто называют абстрактными (не путать с концепцией АТД).

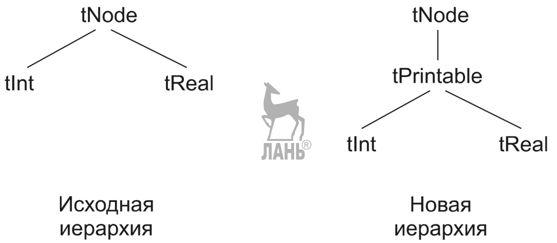


Рис. 1.18. Иерархия типов объектов

Запишем теперь по-новому, в объектном стиле, обслуживание очереди. Но перед этим нужно модернизировать и ее заполнение, поскольку у всех объектов, которые мы записываем в очередь, появилось поле `Print`, которое необходимо задать.

```
(* Заполнение очереди *)
i := 1;
WHILE i<=10 DO
   NEW(int); int.x := i; int.Print := PrintInt;
   Queue.Put(Q, int);
   NEW(real); real.x := i/10; real.Print := PrintReal;
   Queue.Put(Q, real);
   INC(i);
END;
```

При заполнении поля `Print` (выполнении присваивания, подобного `int.Print := PrintInt`) есть риск ошибиться, назначив объекту обработчик сообщения, не соответствующий типу этого объекта. Ведь все обработчики однотипны (в смысле процедурного типа) и компилятор не обнаружит несоответствия, если мы, к примеру, запишем `int.Print := PrintReal`. Риск можно снизить, а заодно сделать программу более структурированной, если для создания объектов, назначения им обработчиков сообщения и заполнения поля значения использовать специальные процедуры, или даже процедуры-функции.

```
PROCEDURE NewInt(i: INTEGER): tInt;
VAR int : tInt;
BEGIN
   NEW(int); int.x := i; int.Print := PrintInt;
   RETURN int
END NewInt;
```

```
PROCEDURE NewReal(r: REAL): tReal;
VAR real : tReal;
BEGIN
   NEW(real); real.x := r; real.Print := PrintReal;
   RETURN real
END NewReal;
```

Используя процедуры-функции `NewInt` и `NewReal`, которые создают объекты в памяти, инициализируют их поля и возвращают в качестве своего значения указатели на созданные объекты, заполнение очереди можно записать компактно и выразительно:

```
(* Заполнение очереди *)
i := 1;
WHILE i<=10 DO
   Queue.Put(Q, NewInt(i));
   Queue.Put(Q, NewReal(i/10));
   INC(i);
END;
```

В языках Объектный Паскаль, Си++, Ява и Си# функции, подобные `NewInt` и `NewReal`, называются конструкторами. Соответственно, в этих языках есть и немало правил, регулирующих использование конструкторов и заодно усложняющих изучение и реализацию языка. В Обероне, где нет понятия «конструктор» (как нет и понятия «объект»), мы, тем не менее, легко такие конструкторы создаем, обходясь обыкновенными процедурами.

И вот — обслуживание очереди в объектном стиле:

```
WHILE Queue.NotEmpty(Q) DO
   Queue.Get(Q, node);
   node(tPrintable).Print(node);
   Out.Ln;
END;
```

Здесь при обращении к полю `Print` объекта `node` использована охрана типа, позволяющая убедиться, что объект имеет динамический тип `tPrintable` (или тип-расширение `tPrintable`), а, следовательно, гарантированно содержит поле `Print`.

Гетерогенная очередь на Обероне-2

Надо сразу сказать, что все программы, написанные в предыдущем разделе на Обероне, являются и правильными программами на Обероне-2. Но язык Оберон-2 содержит связанные процедуры, которые

позволяют несколько по-другому реализовать объектный подход при работе с очередью.

Использование полей процедурного типа для придания объектам индивидуального поведения обеспечивает максимальную гибкость. Значениями таких полей являются процедуры-обработчики сообщений. Разные экземпляры объектов одного и того же типа могут иметь разные значения процедурных полей, а значит, обладать различным поведением. Но такая гибкость часто избыточна. Однотипные объекты и вести себя должны, скорее всего, одинаково. Кроме того, наличие полей процедурного типа у каждого экземпляра объекта увеличивает расход памяти и требует инициализации таких полей каждый раз, когда объект создается, что связано с дополнительными затратами ресурсов, а неверная инициализация или ее отсутствие — потенциальная причина ошибок.

Процедуры-обработчики сообщений могут быть связаны не с экземплярами, а с типами объектов. Подобные процедуры в Обероне-2 и называются «связанными с типом процедурами» (type-bound procedures) или просто «связанными процедурами». В некоторых языках подобные процедуры и функции носят название методов. А типы, с которыми связаны методы, называют классами.

Определим связанные процедуры для типов `tInt` и `tReal`. Названия процедур будут одинаковы — `Print`. Предполагается, что процедурное поле `Print` у типа `tPrintDesc` теперь отсутствует, оно больше не нужно. Роль обработчиков сообщений будут играть связанные с типами процедуры.

```
PROCEDURE (this: tInt) Print;
BEGIN
   Out.Int(this.x, 16)
END Print;

PROCEDURE (this: tReal) Print;
BEGIN
   Out.Real(this.x, 16)
END Print;
```

Отличительным признаком связанных процедур является параметр-приемник[42], описание которого помещается в скобках перед названием процедуры. По типу этого параметра и устанавливается связь про-

[42] Приемник сообщений.

цедуры с типом-записью или типом-указателем на запись. В нашем примере параметром-приемником является `this` — указатель на объект, элемент списка. Внутри связанной процедуры параметр-приемник обозначает тот экземпляр объекта (или указатель на экземпляр объекта), который принимает и обрабатывает сообщение. Отсюда и название, которое я выбрал для параметра-приемника (`this` — этот). В таких языках, как Ява и Си# слово `this` даже является зарезервированным, а соответствующий параметр передается методу неявно. Явная спецификация параметра-приемника в Обероне-2 существенно повышает наглядность программы и облегчает понимание.

Связанные процедуры в некотором отношении похожи на поля записей. Обращение к ним выполняется так же, как и к полям. Существование двух связанных с разными типами процедур, имеющих одинаковое название, разрешено, как разрешено определять одноименные поля внутри разных типов (поля `x` в нашем примере). Но, в отличие от полей, для которых отводится память в каждом экземпляре объекта (переменной-записи), использование связанных процедур не предполагает резервирования памяти для ссылок на них в каждом объекте.

Тип, являющийся расширением другого типа, может иметь связанную процедуру с тем же названием, что имеется у типа-предка. В этом случае типы параметров таких процедур также должны совпадать, а процедура типа-наследника переопределяет соответствующую процедуру типа-предка.

Как и в предыдущих примерах, чтобы запрограммировать обслуживание очереди в объектном стиле, нам потребуется абстрактный промежуточный тип `tPrintable`. Но поскольку для обработки сообщений теперь используются не поля процедурного типа, а связанные процедуры, с этим типом будет связана абстрактная, не выполняющая никакой реальной работы и никогда не вызываемая процедура `Print`.

```
TYPE
   tPrintable = POINTER TO tPrintDesc;
   tPrintDesc = RECORD (Queue.tNodeDesc) END;
   …
PROCEDURE (this: tPrintable) Print;
END Print;
```

Теперь запишем фрагмент, выполняющий заполнение очереди. В отличие от примеров на Обероне, не требуется заполнять поля про-

цедурного типа, которых уже и нет. Кроме того, уместно применить оператор **FOR**, которого нет в Обероне, но есть в Обероне-2.

```
(* Заполнение очереди *)
FOR i := 1 TO 10 DO
   NEW(int); int.x := i;
   Queue.Put(Q, int);
   NEW(real); real.x := i/10;
   Queue.Put(Q, real);
END;
```

Можно, как и раньше, использовать процедуры-конструкторы, которые теперь освобождаются от обязанности заполнять процедурные поля.

И, наконец, можем запрограммировать обслуживание очереди.

```
WHILE Queue.NotEmpty(Q) DO
   Queue.Get(Q, node);        (* Получить        *)
   node(tPrintable).Print;    (* Напечатать      *)
   Out.Ln;                    (* Перевод строки *)
END;
```

Ключевую роль играет здесь оператор `node(tPrintable).Print`. Исполняется он следующим образом. Вначале с помощью охраны типа проверяется, имеет ли указатель `node` динамический тип `tPrintable` или тип, являющийся расширением `tPrintable`. В нашем случае это условие выполнено, поскольку `node` указывает либо на элемент типа `tIntDesc`, либо на элемент типа `tRealDesc`, то есть `node` имеет либо динамический тип `tInt`, либо — `tReal`. И тот, и другой являются расширениями `tPrintable`. Охрана не препятствует дальнейшей работе, поскольку все типы, начиная с `tPrintable`, имеют связанную процедуру `Print`. Далее происходит вызов процедуры `Print`, *связанной с динамическим типом* `node`. То есть, если `node` имеет динамический тип `tInt`, то вызывается `Print`, связанная с `tInt` и умеющая печатать целочисленное значение, если `node` — типа `tReal`, то вызывается связанная с этим типом процедура `Print`.

Такой механизм вызова процедур, когда фактически вызванная процедура становится известна только на этапе выполнения программы, называется *поздним связыванием*. Впрочем, позднее связывание имеет место и при использовании полей процедурного типа. Рассмотрение приведенного фрагмента дает повод упомянуть еще один термин, используемый в разговорах про ООП, — *полиморфизм*. Прояв-

ляется полиморфизм в том, что за некоторыми обозначениями в нашей программе могут скрываться объекты разной природы. Так, в операторе `node(tPrintable).Print` идентификатор `node` может обозначать указатель и на объект с целочисленным значением и на объект с вещественным значением, а если расширить систему типов, то `node` будет соответствовать еще большему набору (без всякого изменения самого оператора). За названием `Print` скрываются в действительности три разные процедуры.

Очередь — тоже объект

Использованный нами объектный подход и связанные процедуры, в частности, можно применить и к модулю, реализующему операции с очередью. Действительно, если элементы очереди мы рассматриваем как объекты, то почему бы и самой очереди не быть объектом? Препятствий для этого нет. Будем считать очередь активным объектом, и вместо того чтоб вызывать, например, процедуру для добавления элемента к очереди, и передавать очередь, в которую надо что-то добавить, в качестве параметра этой процедуре мы будем посылать очереди сообщение «добавь элемент». Она и добавит.

На самом деле, большой разницы в этих двух подходах нет. Ведь передача сообщения — это не что иное, как вызов процедуры. А очередь в качестве параметра этой процедуре все равно передается. Только теперь это будет параметр-приемник. И говорим мы про все немного другими словами.

В листинге 1.12 приведен полный текст модернизированного модуля для работы с очередью.

Листинг 1.12. Модуль для работы с очередью, использующий связанные процедуры

```
MODULE Queue;
TYPE
   tNode* = POINTER TO tNodeDesc;
   tNodeDesc* = RECORD
      Next: tNode
   END;
   tQueue* = RECORD
      First, Last : tNode
   END;

 (* Инициализировать *)
PROCEDURE (VAR Q: tQueue) Init;
```

```
BEGIN
   Q.First := NIL; Q.Last := NIL;
END Init;

(* Добавить элемент *)
PROCEDURE (VAR Q: tQueue) Put*(node: tNode);
BEGIN
   IF Q.First = NIL THEN
      Q.First := node;
   ELSE
      Q.Last.Next := node;
   END;
   Q.Last := node;
   node.Next := NIL;
END Put;

(* Получить элемент *)
PROCEDURE (VAR Q: tQueue) Get*(VAR node: tNode);
BEGIN
   node := Q.First;
   Q.First := Q.First.Next;
   IF Q.First = NIL THEN Q.Last := NIL END;
END Get;

(* Не пусто? *)
PROCEDURE (VAR Q: tQueue) NotEmpty*(): BOOLEAN;
BEGIN
   RETURN Q.First # NIL;
END NotEmpty;

END Queue.
```

Обратите внимание, что в роли приемников связанных процедур используются параметры-переменные типа запись (`tQueue`). Тип указателя на очередь мы не определяем. Если параметр-приемник — запись, а не указатель на запись, в его описании по правилам Оберона-2 должно присутствовать слово **`VAR`**. Поэтому **`VAR`** импользуется даже в процедуре-функции `NotEmpty`, которая не изменяет очередь.

Интерфейс модуля `Queue`, сформированный компилятором JOB, выглядит так:

```
DEFINITION Queue;

TYPE
```

```
   tNode = POINTER TO tNodeDesc;
   tNodeDesc = RECORD
   END;
   tQueue = RECORD
   END;

PROCEDURE ( VAR Q : tQueue ) Init;
PROCEDURE ( VAR Q : tQueue ) Put( node : tNode );
PROCEDURE ( VAR Q : tQueue ) Get( VAR node : tNode );
PROCEDURE ( VAR Q : tQueue ) NotEmpty(  ) : BOOLEAN;

END Queue.
```

Поскольку мы рассматривали много разных вариантов заполнения и обслуживания очереди, цельное представление о том, как должна быть организована программа, работающая с очередью, наверное, оказалось потерянным. Чтобы восстановить это представление привожу полный текст модуля `Test` (листинг 1.13), в котором заполняется и обслуживается (в процедуре `Run`) очередь. При этом используется последняя версия модуля `Queue`.

Листинг 1.13. Использование объектно-ориентированной очереди

```
MODULE Test;
IMPORT Queue, Out;

TYPE
   tPrintable = POINTER TO tPrintDesc;
   tPrintDesc = RECORD (Queue.tNodeDesc) END;

   tInt = POINTER TO tIntDesc;
   tIntDesc = RECORD (tPrintDesc)
      x : INTEGER
   END;

   tReal = POINTER TO tRealDesc;
   tRealDesc = RECORD (tPrintDesc)
      x : REAL
   END;

(* Связанные процедуры *)

   PROCEDURE (this: tPrintable) Print;
   END Print;
```

```
   PROCEDURE  (this: tInt) Print;
   BEGIN
      Out.Int(this.x, 16);
   END Print;

   PROCEDURE  (this: tReal) Print;
   BEGIN
      Out.Real(this.x, 16);
   END Print;

(* Конструкторы *)

   PROCEDURE NewInt(i: INTEGER): tInt;
   VAR int : tInt;
   BEGIN
      NEW(int); int.x := i; RETURN int
   END NewInt;

   PROCEDURE NewReal(r: REAL ) : tReal;
   VAR real : tReal;
   BEGIN
      NEW(real); real.x := r; RETURN real
   END NewReal;

PROCEDURE Run;
VAR
   Q    : Queue.tQueue;
   node : Queue.tNode;
   i    : INTEGER;
BEGIN
    (* Инициализация *)
      Q.Init();
    (* Заполнение *)
      FOR i := 1 TO 10 DO
         Q.Put( NewInt(i) );
         Q.Put( NewReal(i/10) )
      END;
    (* Обслуживание *)
      WHILE Q.NotEmpty() DO
         Q.Get(node);
         node(tPrintable).Print;
         Out.Ln
      END;
END Run;

END Test.
```

Гетерогенная очередь на языке Ява

Основные идеи, которые будут использованы при реализации и иллюстрации использования очереди на языке Ява те же, что и при использовании языка Оберон-2. По сути, это будут те же программы, но отличающиеся формой записи и некоторыми деталями. К сожалению, в Яве отсутствует понятие модуля, поэтому программы представляют собой «россыпь» зачастую довольно мелких классов. Классы в Яве — аналог записей со связанными процедурами-методами. Классы, содержащие статические члены, похожи на модули.

Начнем с классов, реализующих очередь. Вначале запишем класс, определяющий тип элементов очереди.

```
public class Node {
   Node next;
}
```

Этот класс должен располагаться в файле `Node.java`. В примерах на Яве я не использую «венгерскую нотацию», когда природа объекта программы подчеркивается с помощью префикса в его имени, а придерживаюсь рекомендаций по выбору имен, приведенных в спецификации (!) языка Ява. Имена классов начинаются с заглавной буквы, имена полей, методов и локальных переменных — со строчной. Имена классов, полей и переменных — существительные, методов — в основном глаголы.

Класс `Queue` (листинг 1.14) отвечает за операции с очередью и располагается в файле `Queue.java`.

Листинг 1.14. Очередь на языке Ява

```
public class Queue {

   private Node first; // Эти поля недоступны
   private Node last;  // другим классам

   // Добавить
   public void put(Node node) {
      if( first == null )
         first = node;
      else
         last.next = node;
      last = node;
      node.next = null;
   }
```

```
   // Получить
   public Node get() {
      Node temp = first;
      first = first.next;
      if( first == null ) last = null;
      return temp;
   }

   // Очередь не пуста
   public boolean notEmpty() {
      return first != null;
   }
}
```

Можно обратить внимание, что отсутствует метод, выполняющий инициализацию очереди. В нем нет необходимости, поскольку при создании объекта (экземпляра очереди) полям этого объекта будут присвоены значения по умолчанию. Для `first` и `last` это будут значения `null`. Такое присваивание выполняется конструктором, который создается неявно, если программист не предусмотрел собственного конструктора для данного класса.

В отличие от варианта на Обероне, метод `get` не имеет выходного параметра, а является функцией типа `Node`, поскольку выходные параметры (или параметры, передаваемые по ссылке) по странному стечению обстоятельств в Яве вообще не предусмотрены.

Теперь рассмотрим варианты заполнения и обслуживания очереди, в которую заносятся целые и вещественные числа. Иерархия классов, в которую входят и классы элементов, образующих очередь, может быть такой же, как и в примерах, записанных на Обероне (листинг 1.15). Наследником `Node` будет класс `Printable`, а от него наследуют `Int` и `Real`. Наследование классов в Яве обозначается словом `extends`.

Листинг 1.15. Классы для элементов очереди

```
abstract class Printable extends Node {
   abstract void print();
}

class Int extends Printable {
   int x;          // Поле
   void print() { // Метод
```

```
      System.out.println(x);
   }
}

class Real extends Printable {
   float x;       // Поле
   void print() { // Метод
      System.out.println(x);
   }
}
```

В Яве абстрактные классы и методы могут быть явно помечены словом `abstract`, что можно увидеть в описании класса `Printable`. Применение явно отмеченных абстрактных классов и методов обеспечивает дополнительные возможности для контроля и экономит ресурсы. Невозможно создать объект абстрактного класса или вызвать абстрактный метод. Код для абстрактных методов даже не создается компилятором. Если в (не абстрактном) классе, наследнике абстрактного, не будут переопределены абстрактные методы, это приведет к ошибке при компиляции.

Хотя методы `print` классов `Int` и `Real` выглядят одинаково (оба содержат только вызов `System.out.println(x)`) , в действительности они различны. Имя метода в Яве не определяет его однозначно (речь в данном случае идет о `println`). Необходимо еще знать типы параметров и возвращаемого значения. Поскольку в одном случае указан параметр `x` типа `int`, а в другом — типа `float`, перед нами вызовы двух различных (совместно используемых) методов `println`.

Теперь можно записать класс `Test`, в котором заполняется и обслуживается очередь (листинг 1.16). Все действия сконцентрированы в методе `main`. С его вызова начинается выполнение программы на языке Ява.

Листинг 1.16. Заполнение и обслуживание очереди

```
public class Test {
   public static void main(String[] args) {
      Queue q = new Queue();
      Int integer;
      Real real;
      Node node;

      // Заполнение
      for( int i = 1; i<=10; i++ ){
```

```
            integer = new Int(); integer.x = i;
            q.put(integer);
            real = new Real(); real.x = i/10f;
            q.put(real);
         }

         // Обслуживание
         while( q.notEmpty() ) {
            node = q.get();
            ((Printable)node).print();
         }

      }

   }
```

Создание очередного объекта выполняется с помощью операции `new` и вызова конструктора. Имена конструкторов в Яве всегда совпадают с названием соответствующего класса. Значением операции `new` является объект этого класса[43]. Поскольку конструкторы мы сами не определили, вызываются конструкторы, создаваемые по умолчанию.

Требует некоторых пояснений оператор `((Printable)node).print();` из цикла обслуживания очереди. Здесь использована операция приведения, обозначенная именем класса `Printable`, заключенным в скобки и записанным перед обозначением переменной `node`. Эта операция преобразует (если возможно) переменную `node` к типу `Printable`. Действие операции приведения в нашем примере абсолютно идентично охране типа в Обероне. После приведения происходит (полиморфный) вызов метода `print` того класса, к какому относится объект, на который ссылается `node`. Дополнительные скобки вокруг `(Printable)node` требуются в связи с действующим в Яве приоритетом операций: приведение имеет меньший приоритет, чем доступ к полю объекта, обозначаемый точкой.

Конструкторы и интерфейсы

Как и в программе на Обероне, использование конструкторов «собственного изготовления» в этом примере вполне уместно. Тем более что понятие «конструктор» предусмотрено самим языком Ява, а их

---

[43] На самом деле речь, конечно, идет об указателе на объект, но в терминологии языка Ява это принято почему-то маскировать.

использование — обычная практика программирования на этом языке.

Вместо абстрактного класса `Printable` можно применить интерфейс `Printable`[44]. Интерфейсы во многом эквивалентны абстрактным классам. Но класс может быть непосредственным потомком единственного суперкласса, а суперинтерфейсов у данного класса может быть несколько. Говорят, что классы не наследуют от интерфейсов, а реализуют их, что изображается в описании класса словом `implements`. При использовании интерфейса `Printable` изменится схема наследования. Классы `Int` и `Real` будут потомками непосредственно `Node` и оба будут реализовывать интерфейс `Printable`.

При использовании конструкторов изменится цикл заполнения очереди (листинг 1.17). В цикле обслуживания очереди сократим запись, воспользовавшись тем, что обращение к методу объекта можно выполнять не только с помощью переменной (как в Обероне), но с помощью любого выражения, дающего ссылку на объект. Описание интерфейса `Printable`, классов `Int` и `Real` и класса `Test` могут размещаться в одном файле, который должен называться `Test.java`.

Листинг 1.17. Использование гетерогенной очереди на Яве

```
public class Test {
   public static void main(String[] args) {
      Queue q = new Queue();
      // Заполнение
         for( int i=1; i<=10; i++ ) {
            q.put(new Int(i));
            q.put(new Real(i/10f));
         }
      // Обслуживание
         while( q.notEmpty() )
            ((Printable)q.get()).print();
   }
}
interface Printable {
   void print();
}
```

[44] Название Printable, использованное в наших примерах с самого начала, как раз соответствует традиции именования интерфейсов при программировании на Яве.

```
class Int extends Node implements Printable {
   int x;                    // Поле
   Int(int i) { x = i; }  // Конструктор
   public void print() {  // Метод
      System.out.println(x);
   }
}

class Real extends Node implements Printable {
   float x;                    // Поле
   Real(float r) { x = r; } // Конструктор
   public void print() {    // Метод
      System.out.println(x);
   }
}
```

Гетерогенная очередь на языке Си#

Рассмотренные выше программы на Яве могут с минимальными изменениями быть преобразованы в программы на языке Си#. Приведу без особенных пояснений аналог последнего варианта, использующего конструкторы и интерфейс.

Разумно разместить программу в двух файлах. Один, `Queue.cs`, содержит все, что относится к реализации очереди (листинг 1.18): класс `Node` и класс `Queue`.

Листинг 1.18. Реализация очереди на языке Си#

```
// Элемент очереди
public class Node {
   internal Node next;
}

// Очередь
public class Queue {

   private Node first;
   private Node last;

   // Добавить
   public void put(Node node) {
      if( first == null )
         first = node;
      else
         last.next = node;
```

```
        last = node;
        node.next = null;
    }

    // Получить
    public Node get() {
        Node temp = first;
        first = first.next;
        if( first == null ) last = null;
        return temp;
    }

    // Очередь не пуста
    public bool notEmpty() {
        return first != null;
    }
}
```

В другом файле, `Test.cs`, размещаются классы, иллюстрирующие использование очереди (листинг 1.19).

Листинг 1.19. Использование очереди

```
public class Test {
    public static void Main() {
        Queue q = new Queue();
        // Заполнение
            for( int i=1; i<=10; i++ ) {
                q.put(new Int(i));
                q.put(new Real(i/10f));
            }
        // Обслуживание
            while (q.notEmpty())
                ((Printable)q.get()).print();
    }
}

interface Printable {
    void print();
}

class Int : Node,  Printable {
    int x;                            // Поле
    public Int(int i) { x = i; }      // Конструктор
    public void print() {             // Метод
```

```
        System.Console.WriteLine(x);
    }
}

class Real : Node, Printable {
    float x;                             // Поле
    public Real(float r) { x = r; } // Конструктор
    public void print() {               // Метод
        System.Console.WriteLine(x);
    }
}
```

Отличия этого текста от соответствующей программы на Яве предлагаю читателям найти самостоятельно.

Использование общего суперкласса для организации очереди

Программируя на Си#, впрочем, как и на Яве, очередь с разнотипными элементами можно организовать и по-другому. В этих языках все классы происходят от одного общего суперкласса, который в Си# обозначается как `object`, а в Яве — `Object`. В дальнейшем буду вести изложение применительно к Си#. Если переменная или поле имеют тип `object`, то они могут ссылаться на объект любого класса, поскольку любой класс является производным (расширением) от `object`. В элементе списка можно хранить не сами данные, а ссылку на них (рис. 1.19).

В этом случае объекты, помещаемые в очередь, не обязаны иметь тип, являющийся производным от `Node`, и программист уже не должен привязывать свою иерархию классов к классу `Node`. В такую очередь можно заносить объекты любого типа.

Реализация очереди немного меняется (листинг 1.20). В классе `Node` появляется новое внутреннее поле `data`. Метод `put` теперь сам создает новый элемент списка, а значением поля `data` этого элемента становится ссылка на добавленный объект. Метод `get` возвращает не указатель на первый элемент очереди, а значение поля `data` этого элемента. Параметр метода `put` и возвращаемое значение метода `get` теперь имеют тип `object`.

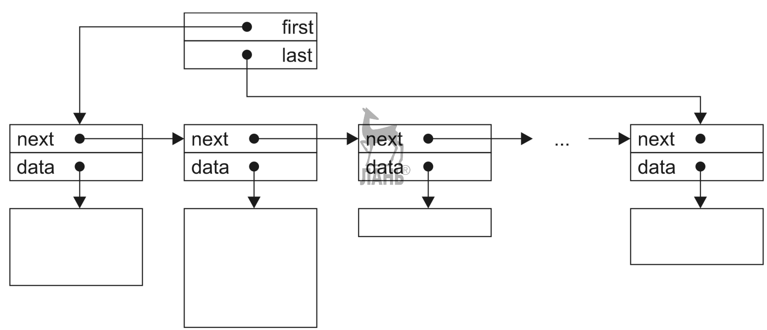


Рис. 1.19. Очередь со ссылками на данные

Листинг 1.20. Очередь со ссылками на данные

```
// Элемент очереди
public class Node {
   internal Node next;   // Поле связи
   internal object data; // Ссылка на данные
}

// Очередь
public class Queue {

   private Node first;
   private Node last;

   // Добавить
   public void put(object data) {
      Node node = new Node(); // Создание элемента
      node.data = data;       // "Привязка" данных
      if( first == null )
         first = node;
      else
         last.next = node;
      last = node;
      node.next = null;
   }

   // Получить
   public object get() {
      Node temp = first;
      first = first.next;
```

```
        if( first == null ) last = null;
        return temp.data;
    }

    // Очередь не пуста
    public bool notEmpty() {
        return first != null;
    }
}
```

Интересно, что с модифицированной реализацией очереди (листинг 1.20) будет без всяких изменений работать прежний вариант тестовой программы (листинг 1.19). Но наследование от `Node` теперь не нужно. Классы объектов, добавляемых в очередь, также можно определить по-новому (из списка наследования исчезает класс `Node`).

```
class Int : Printable {
    int x;                              // Поле
    public Int(int i) { x = i; }        // Конструктор
    public void print() {               // Метод
        System.Console.WriteLine(x);
    }
}

class Real : Printable {
    float x;                            // Поле
    public Real(float r) { x = r; }     // Конструктор
    public void print() {               // Метод
        System.Console.WriteLine(x);
    }
}
```

Использование упаковки и распаковки

В языке Си# есть возможность, отличающая его от языка Ява[45] и других языков и позволяющая нам рассмотреть еще один вариант использования очереди. Эта возможность связана с упаковкой (boxing) и распаковкой (unboxing).

Переменной типа `object` можно присвоить значение не только ссылочного типа (класс, интерфейс, массив), но также и любого при-

[45] В Яве предусмотрены так называемые классы-обертки (wrapper classes), использование которых в некотором отношении напоминает применение упаковки и распаковки. В последних версиях языка Ява предусмотрены автоматическая упаковка и распаковка.

митивного типа (`int`, `float` и др.), а также значение структуры. При этом происходит неявная упаковка величины такого типа-значения в объект-оболочку. То есть создается новый объект, единственным полем которого является поле, хранящее упакованное значение. Как любой объект, он оказывается совместим по присваиванию с типом `object`. Обратное преобразование (из ссылочного типа к типу-значению), называемое распаковкой, должно выполняться явно с помощью операции приведения.

Использование упаковки в нашем примере позволяет при занесении данных в очередь обойтись без вызова конструкторов и операции `new`, записывая вместо `q.put(new Int(i));` просто `q.put(i);`. Объект, который заносится в очередь, будет создан автоматически в результате упаковки. Правда, типом этого объекта не будут `Int` или `Real`[46], а, следовательно, приведение к `Printable` в операторе `((Printable)q.get()).print();` при обслуживании очереди приведет к ошибке.

Не имея возможности привести тип объектов, получаемых из очереди к `Printable`, мы не можем воспользоваться и методами `print`. Более того, классы `Int` и `Real`, содержащие такие методы, теперь вообще не имеют никакого отношения к типу объектов в очереди. Можно осуществить обслуживание очереди, проверяя в цикле тип каждого объекта.

```
while( q.notEmpty() ) {
   node = q.get();
   if( node is int )
      System.Console.WriteLine((int)node);
   else
      System.Console.WriteLine((float)node);
}
```

Предполагается, что вспомогательная переменная `node` имеет тип `object`. Обратите внимание на операцию проверки динамического типа. В Си# у нее такое же обозначение, как и в Обероне — `is`. Распаковка выполняется в этом примере в выражениях `(int)node` и `(float)node`.

[46] Такой объект по правилам Си# будет иметь динамический (ссылочный!) тип `int` или `float`.

Однако можно напечатать значения элементов очереди и не прибегая к проверке типа в цикле обслуживания и не нарушая тем самым принципы объектного подхода (листинг 1.21). Для печати достаточно использовать `System.Console.WriteLine`. Получается самый компактный вариант использования очереди.

Листинг 1.21. Применение упаковки для заполнения очереди

```
public class Test {
   public static void Main() {
      Queue q = new Queue();
      // Заполнение
         for (int i=1; i<=10; i++ ) {
            q.put(i);
            q.put(i/10f);
         }
      // Обслуживание
         while (q.notEmpty())
            System.Console.WriteLine(q.get());
   }
}
```

Преобразование значения `q.get()`, которое может быть различного типа, в последовательность символов, выводимых на печать, выполняется методом `WriteLine`. Он, в свою очередь, обращается к методу `ToString` соответствующего объекта. Для объектов, полученных упаковкой стандартных примитивных типов, таких как `int` и `float` в нашем примере, метод `ToString` реализован в стандартной библиотеке Си#.

Языки-концепции

Рассмотренные в предыдущих разделах языки можно считать традиционными. Несмотря на значительные отличия друг от друга, в их основе много общего. Все они, так или иначе, происходят от Фортрана и Алгола-60.

Конечно, немало новых идей воплотилось в этих языках: структурное, модульное, объектно-ориентированное программирование. Но такие базовые понятия-конструкции как переменная, выражение, оператор, сохранились неизменными по сути и почти неизменными по форме. Кроме этого, обсуждавшиеся языки объединены своей императивной природой — программа есть указание последовательности выполняемых действий.

Следование традиции обеспечивает упомянутым языкам самое широкое распространение[47]. Именно с их помощью создается подавляющее большинство современных программ.

Вместе с тем существовали и существуют гораздо менее распространенные языки программирования, совсем не традиционные, но от этого не менее значимые. Каждый из таких языков, как правило, воплощает какую-то оригинальную идею, доказывая возможность построения универсального языка, использующего, например, только одну структуру данных или, допустим, единственную программную сущность — объекты.

Знакомство с такими языками-концепциями следует считать безусловно полезным, но, к сожалению, формат этой книги и мои возможности не позволяют рассказать о них подробно. Поэтому ограничусь короткими справками.

Форт

Язык Форт (Forth), кроме того, что интересен сам по себе, важен для нас еще и потому, что его основная концепция и система обозначений будут использоваться в последующих главах при обсуждении методов трансляции.

Форт разработан в начале 1970-х годов американцем Чарльзом Муром (Charles Moore). Мур применил созданный им язык для программирования системы управления радиотелескопом в Национальной Радиоастрономической обсерватории США. Название языка происходит от английского forth — «вперед» и одновременно является производным от fourth — «четвертый» (Мур считал, что язык станет средством программирования компьютеров четвертого поколения).

Основные черты языка Форт

- Единственная структура данных — стек. Операнды всех операций находятся в стеке, результат помещается в стек.
- Основной тип данных — целый. Более поздние реализации Форта предусматривают работу и с вещественными числами.

[47] Речь идет, конечно, о совокупной распространенности. Си и Си++, пожалуй, применяются больше всего. Растет популярность языка Ява. Однако Алгол-60 сейчас не используется совсем. Почти забыт ПЛ/1. Не имеет широкого распространения Оберон.

- Постфиксная запись программы — обозначение операции записывается после операндов. Такую запись называют «обратной польской».
- Предельно простой синтаксис, компактная запись.
- Малая потребность в ресурсах, высокая эффективность. Форт реализуется с помощью интерпретатора, который имеет и черты компилятора. Интерпретация основывается на использовании так называемого «шитого кода». Быстродействие Форт-программы оказывается лишь ненамного меньше соответствующей программы в машинном коде.
- Отсутствие контроля типов и контроля во время выполнения. Незащищенность Форт-системы от ошибок программиста.
- Трудность чтения текста программы из-за ее компактной записи в обратной польской форме и широко применяемых неочевидных манипуляций со стеком.

Примеры программирования на Форте

Основной единицей Форт программы является слово. Программа — последовательность слов. В качестве слова рассматривается любая последовательность знаков, не включающая пробела. Имеются предопределенные слова, выполняющие заранее определенные действия.

Действие слова, представляющего собой запись целого числа без знака, состоит в том, что это число заносится в стек (на вершину стека).

Слово «.» (точка) выводит число с вершины стека на терминал. Число из стека при этом удаляется.

Работа с Форт-системой может происходить в режиме диалога. Слова, вводимые в ответ на приглашение системы, немедленно исполняются. В следующем примере диалога с Форт-системой числа 10 и 20 вначале заносится в стек, а затем выводятся с помощью слова «.». Стек после этого становится пустым. В качестве приглашения системы используется знак «>». Курсивом отмечен ввод пользователя, прямым шрифтом — ответы системы.

```
>10 20
>. .
20 10
```

Понятно, что числа выведены в порядке обратном вводу, поскольку для стека действует правило «последним пришел — первым ушел».

Рассмотрим еще несколько стандартных слов Форта. Их обозначения и действие приведены в таблице 1.1.

Таблица 1.1. Некоторые стандартные слова Форта

Слово	Действие	Стек
+	Заменяет два верхних элемента стека их суммой	A, B → A+B
–	Заменяет два верхних элемента стека их разностью (верхний элемент вычитается из нижнего)	A, B → A–B
*	Заменяет два верхних элемента стека их произведением	A, B → A*B
/	Заменяет два верхних элемента стека их частным	A, B → A/B
DUP	Дублирует содержимое вершины стека	A → A, A
DROP	Удаляет верхний элемент стека	A →
SWAP	Меняет местами два верхних элемента стека	A, B → B, A

В графе «Стек» использовано традиционное для описаний Форта обозначение действий со стеком. Слева от стрелки показано состояние верхних элементов стека до выполнения действия (слова, команды), справа от стрелки — после. Вершина стека предполагается расположенной справа. У нас будет повод использовать такую нотацию и в дальнейшем.

Продолжим упражнения в программировании на Форте. Попробуйте определить, какой результат даст следующая последовательность слов-команд:

```
>5 5 * 4 4 * + .
```

Что будет напечатано словом «точка», которое выполняется последним? 41? Правильно! В самом деле, вначале первое слово 5 заносит константу 5 в стек. Второе такое же слово помещает поверх пятерки еще одну. Слово «*» заменяет два верхних элемента стека их произведением — две пятерки заменяются на 25. Далее 4 таким же способом умножается на 4, получившееся число 16 оказывается на вершине стека поверх числа 25. Наконец, слово «+» заменяет 25 и 16 их суммой, записывая в стек 41. А слово «.» снимает 41 со стека и

выводит его на терминал. Стек остается пустым (точнее, остается в том же состоянии, в котором был до выполнения вычислений).

Форт позволяет определять собственные слова, а затем использовать их наравне со стандартными. Определим, например, слово S2, которое позволит вычислять сумму квадратов двух чисел, находящихся на вершине стека. Для этого используем так называемое «определение через двоеточие»:

```
>: S2 DUP * SWAP DUP * + ;
```

За словом «:» (двоеточие) записываем определяемое слово (S2), затем команды (слова), обеспечивающие нужное вычисление. Заканчивается определение через двоеточие словом «;». После ввода такой строки новое слово S2 добавляется к словарю Форт-системы. Его можно применять. Например, так:

```
>4 5 S2 .
41
```

Роль Форта

На рубеже 1970-х и 1980-х годов Форт был весьма популярен. На первых персональных компьютерах нетребовательные к ресурсам системы программирования на Форте были реализованы одними из первых. Используется Форт и теперь. Известно немало его реализаций, в том числе отечественных.

Форт можно считать определенной альтернативой языку ассемблера. Незначительно уступая в эффективности, обладая большой гибкостью, он привлекает машинной независимостью и возможностями саморасширения.

Лисп

Язык программирования Лисп (Lisp — от List Processing — обработка списков) предложен сотрудником Массачусетского технологического института Джоном Маккарти (John McCarthy) в начале 1960-х годов. Язык возник в ходе исследований по искусственному интеллекту и символьным вычислениям. В этой сфере Лисп и его диалекты сохраняют свое значение и поныне.

Основная структура данных Лиспа — связанный список. Программы также представлены в виде списков. Лисп относится к числу языков *функционального программирования*, в которых основным дей-

ствием является вызов функции. Важную роль играет рекурсия, заменяющая циклы.

Существует большое число диалектов Лиспа, некоторые из которых приобрели статус самостоятельных языков. Это язык Схема (Scheme, 1975), используемый в учебных курсах по функциональному программированию, язык Common Lisp (1984) — расширенный и стандартизованный диалект Лиспа, язык CLOS (Common Lisp Object System) — расширенный средствами объектно-ориентированного программирования Common Lisp.

Пролог

От сокращения слов «логическое программирование» (programming in logic) происходит название языка Пролог (Prolog). Пролог — неимперативный язык. Программа на Прологе задает не последовательность действий, которые должны быть совершены, а определяет совокупность правил и фактов, из которых решение выводится встроенной в систему программирования на Прологе «машиной вывода».

Пролог возник в ходе исследований в области искусственного интеллекта как система для доказательства теорем. Разработка была выполнена на Фортране группой специалистов Марсельского университета под руководством Алана Колмероэ (Alain Colmerauer) в начале 1970-х годов.

В конце 1970-х и в 80-е годы можно было наблюдать немалый энтузиазм в отношении Пролога. Его использование связывалось с получившими тогда известность экспертными системами.

Большой переполох наделал широко разрекламированный в начале 1980-х японский проект создания компьютеров «пятого поколения», получивший даже название «японский вызов». Базовым языком этого проекта был назван Пролог. Однако очевидный неуспех проекта убавил популярности и Прологу.

Смолток

Смолток (Smalltalk) — это, без сомнения, самый объектно-ориентированный из всех объектно-ориентированных языков программирования. Объекты и ничего кроме объектов — таков принцип Смолтока. В форме объектов представляется всё. Даже обычные числа. Чтобы выполнить, например, сложение одно число-объект посылает сообщение другому числу.

Смолток был создан в начале 1970-х годов в Исследовательском центре компании Ксерокс в Пало Альто (Xerox PARC, Xerox Palo Alto Research Center) в рамках пионерского во многих отношениях проекта, целью которого была разработка мощного, компактного и простого в использовании компьютера. Основные принципы построения такой системы, получившей название «Дайнабук» (Dynabook— динамическая книга), были сформулированы Аланом Кеем (Alan Kay). В проекте Дайнабук в 1973 году был впервые реализован оконный графический пользовательский интерфейс, использующий меню и манипулятор «мышь».

Смолток составлял важную часть системы, представляя собой не только язык, но и визуальную среду программирования. Вклад в создание, реализацию и развитие языка кроме А. Кея внесли Дэн Ингалс (Dan Ingalls), Адель Гольдберг (Adele Goldberg) и другие сотрудники исследовательской группы Xerox PARC. Были выпущены несколько версий: Смолток-72, -74, -76, -78, -80. Смолток-80 был перенесен на многие платформы и получил значительное распространение. На конструкцию Смолтока оказали влияние языки Симула-67, Лого — язык, предназначенный для обучения программированию на графических примерах, и FLEX — более ранний язык, предложенный Кеем.

Язык и система программирования Смолток оказали значительное влияние на последующее развитие. В первую очередь это относится к графическому интерфейсу пользователя. Принципы, впервые реализованные в Xerox PARC, были затем применены Н. Виртом на компьютерах Lilith и Ceres, послуживших платформой для языков Модула-2 и Оберон, компанией Apple при создании операционной системы с графическим интерфейсом для компьютеров Macintosh и, наконец, компанией Microsoft в системе Windows. Реализуя идеи объектно-ориентированного программирования в чистом виде, Смолток служит убедительным доказательством их продуктивности. Черты Смолтока можно найти в более поздних объектно-ориентированных языках, например, в Яве.

Существовали и проблемы в использовании и распространении Смолтока. Дело в том, что механизмы, которые лежат в основе системы (динамическое распределение памяти, позднее связывание, проверка типов во время выполнения, автоматическая сборка мусора), порождают неэффективность. Для работы с приемлемой скоростью Смолток-системе требовался компьютер с гораздо большими

ресурсами, чем имевшиеся на массовых машинах конца 1970-х — начала 1980-х годов.

Подобные трудности были характерны и для таких концептуальных языков, как Лисп и Пролог. Проблема практически исчезла лишь к началу XXI века, когда мощность персональных компьютеров неизмеримо возросла, а использование упомянутых механизмов стало повсеместным в таких языках как Оберон, Ява, Си#.

Языки Интернета

Массовое распространение Интернета в середине 1990-х годов заметно повлияло и на ситуацию с языками программирования. Для создания программного обеспечения интернет-сайтов, других программ, работающих в сети и с сетью, используются как традиционные языки программирования, обсуждавшиеся в предыдущих разделах, так и языки, специально разработанные для тех или иных задач сетевого программирования.

HTML

Собственно, Интернет, точнее самая популярная его подсистема WWW (World Wide Web — Всемирная Паутина), основан на использовании языка гипертекстовой разметки HTML — Hyper Text Markup Language. Вот пример описания на языке HTML простейшей веб-страницы.

Листинг 1.22. HTML-страница «Hello, World!»

```
<HTML>
   <HEAD>
      <TITLE>Простейшая HTML-страница</TITLE>
   </HEAD>
   <BODY>
   Hello, World!
   </BODY>
</HTML>
```

То, что будет видно при просмотре этой страницы в браузере, можно наблюдать на рисунке 1.20.

Элементы страницы задаются тегами. В примере использованы теги `<HTML>`, `<HEAD>`, `<TITLE>`, `<BODY>`. Все они завершаются явно конструкциями `</HTML>`, `</HEAD>`, `</TITLE>`, `</BODY>` соответственно. Могут быть и теги, не предусматривающие явного завершения.

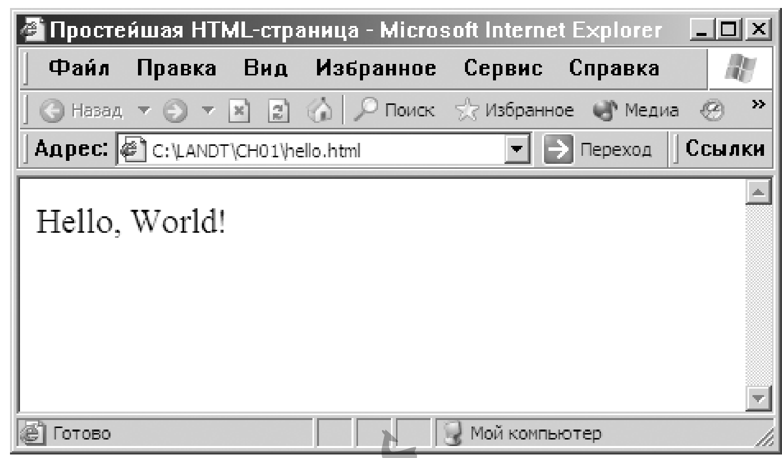


Рис. 1.20. HTML-страница «Hello, World!» в окне браузера

HTML вряд ли следует считать языком программирования, это язык описания страниц. Сам по себе HTML не содержит средств задания действий, динамики, поведения. Динамическое содержание привносится на веб-страницы программами, написанными на языках, отличных от HTML.

Ява и аплеты

Одна из первых возможностей оснащения страниц Интернета динамическими элементами была связана с появлением языка Ява в 1995 году. Небольшие программы-аплеты, написанные на Яве, компилируются в независящий от платформы байт-код, который размещается в виде файлов классов на веб-сервере. Байт-код сконструирован так, что размер файлов классов относительно невелик и их передача по сети не требует слишком большого времени. На веб-страницах размещаются ссылки на файлы классов в виде специальных конструкций-тегов языка HTML (тег `<applet>`). При загрузке HTML-страницы браузер считывает с сервера файлы классов тех аплетов, на которые имеются ссылки на странице. Выполняются аплеты на компьютере клиента (посетителя веб-страницы) с помощью встроенной в браузер виртуальной Ява-машины (JVM — Java Virtual Machine). В ходе выполнения аплета составляющие его классы

загружаются через сеть динамически, то есть в тот момент, когда происходит обращение к какому-либо ресурсу, экспортированному классом. В комплект JVM входит обширная библиотека классов, обеспечивающих графический вывод, взаимодействие с пользователем, оконной и файловой системой, сервером. Наличие этой библиотеки на компьютере клиента позволяет уменьшить объем загружаемых по сети данных.

С помощью Ява-аплетов можно решать разнообразные задачи. Технология весьма универсальна. Однако, как показала практика, аплеты не стали самым распространенным способом доставки на веб-страницы динамического содержания. По-видимому, это объясняется усложненной технологией изготовления аплетов, требующей хорошего знания непростого языка Ява, предполагающей отдельный этап компляции программы с получением и последующим размещением на сервере многочисленных файлов классов. Сказалось, вероятно, и наличие ошибок в первых реализациях виртуальной машины языка Ява и библиотеки классов, создававшее немалые проблемы при отладке и использовании аплетов.

Как уже отмечалось, технология Ява-аплетов к 2010-м годам почти перестала использоваться, а начиная с 2015 года, поддержка аплетов в некоторых браузерах совсем прекратилась.

Файлы классов, способные исполняться на JVM, могут быть получены трансляцией не только с языка Ява. Существует компиляторы для многих языков программирования, позволяющие получать байт-код и программировать аплеты. Примером может служить компилятор JOB, преобразующий программы на языке Оберон-2 в файлы классов. Надо, однако, сказать, что устройство JVM ориентировано именно на язык Ява и трансляция в байт-код некоторых конструкций других языков вызывает определенные трудности.

Была предложена и альтернативная технология аплетов. На базе языка Оберон учеником Н. Вирта профессором Калифорнийского университета в Ирвине Михаэлем Францем (Michael Franz) была создана технология Juice. Она предусматривала трансляцию написанных на Обероне аплетов в специальное, более компактное, чем байт-код Явы промежуточное представление с последующей трансляцией в машинный код на компьютере клиента переданных по сети файлов промежуточного кода. Несмотря на ряд преимуществ перед Ява-технологией, Juice осталась экспериментальной разработкой.

Скриптовые языки

Заметно чаще, чем аплеты на интернет-страницах используются скрипты — программы, исходный текст которых включается прямо в HTML-страницу и исполняется встроенным в браузер интерпретатором.

Самым распространеным скриптовым языком стал Java Script, поддержка которого была впервые включена в браузер компании Netscape. Вопреки распространенному заблуждению Java Script не является диалектом языка Ява. В языке можно увидеть черты Си, Паскаля, есть средства объектно-ориентированного программирования.

В силу тесной интеграции с браузером, программа на Java Script имеет доступ практически к любой информации о текущем состоянии активной страницы и событиях, происходящих при ее просмотре. Благодаря этому возможности скриптов достаточно широки. На JavaScript (как и на Яве, конечно) можно, например, запрограммировать игру, работающую на веб-странице. В приведенном ниже листинге 1.23 можно видеть простейший пример использования JavaScript.

Листинг 1.23. «Hello, World!» на JavaScript

```
<HTML>
   <HEAD>
      <TITLE>Страница со скриптом на JavaScript</TITLE>
   </HEAD>
   <BODY>
      <SCRIPT Language="JavaScript">
         document.write("Hello, World!")
      </SCRIPT>
   </BODY>
</HTML>
```

При загрузке этой страницы в браузер мы увидим то же, что и в предыдущем примере (см. рис. 1.20). Браузер получит полный текст страницы и исполнит скрипт. Использование JavaScript в этом примере не приносит никакой пользы, а пример дан лишь для того, чтоб показать, каким образом скрипт внедряется в HTML-код с помощью тега `<SCRIPT>`. Нетрудно, однако, представить, что скрипт предоставляет массу возможностей: использование циклов, вывод в зависимости от условий разных текстов, диалоговые окна и т. д.

Компанией Microsoft для тех же целей, что и JavaScript предложен диалект языка Visual Basic под названием VB Script (листинг 1.24).

Листинг 1.24. «Hello, World!» на VB Script

```
<HTML>
   <HEAD>
      <TITLE>Страница со скриптом на VB Script</TITLE>
   </HEAD>
   <BODY>
      <SCRIPT Language="VBScript">
         Document.Write "Hello, World!"
      </SCRIPT>
   </BODY>
</HTML>
```

Как видно, отличия от JavaScript в этом примере минимальны. Разве что либеральный синтаксис VB Script позволяет не записывать скобки там, где в JavaScript они обязательны. Ну, а результат, конечно, будет такой же (см. рис. 1.20).

Популярность технологии скриптов объясняется, по-видимому, предельной простотой ее использования. Чтобы задействовать скрипт, программисту не нужны никакие дополнительные инструменты (компиляторы и т. п.). Поведение записанного на странице скрипта можно тут же наблюдать в браузере. Кроме того, поскольку исходные тексты скриптов всегда открыты (в отличие от текстов аплетов), существует масса примеров, которые можно изучать и использовать.

Языки CGI-программирования

Аплеты и скрипты, исполняемые на компьютере клиента-посетителя веб-страницы, ограничены в своих правах доступа к файлам веб-сервера. В то время как программа, работающая на сервере, может читать и записывать расположенные там файлы, выполняя запросы к размещенным на сервере базам данных, обновляя эти базы и направляя результаты на веб-страницу. Порядок взаимодействия браузера (веб-страницы) с выполняемой на веб-сервере программой определяется интерфейсом CGI (Common Gateway Interface). В роли программы, взаимодействующей с помощью CGI-интерфейса с веб-сервером и веб-браузером, может, в принципе, выступать любая программа, способная исполняться в среде той операционной системы, на которой работает веб-сервер. И написана она может быть практически на любом языке. Для Unix-подобных ОС это будет, к примеру, программа, написанная на Си и откомпилированная в машинный код

компилятором языка Си, который входит в состав любой такой системы. Занимаясь CGI-программированием для веб-сервера, работавшего под управлением ОС Windows, я использовал языки Паскаль и Оберон, создавая исполнимые (.exe) файлы с помощью компиляторов Delphi и XDS. *Для CGI-программирования никакого специального языка не требуется*.

Результат своей работы, направляемый по сети браузеру, CGI-программа может просто выводить в стандартный выходной файл. Это видно в листинге 1.25, где приведена программа на Паскале, формирующая для браузера HTML-страницу «Hello, World!».

Листинг 1.25. CGI-программа «Hello, World!»

```
program Hello;
begin
   WriteLn('Content-type text/html');
   WriteLn;
   WriteLn('<HTML>');
   WriteLn(' <HEAD>');
   WriteLn(' <TITLE>Привет от CGI-программы</TITLE>');
   WriteLn(' </HEAD>');
   WriteLn(' <BODY>');
   WriteLn(' Hello, World!');
   WriteLn(' </BODY>');
   WriteLn('</HTML>')
end.
```

При выполнении на сервере этой программы выводимая ею HTML-страница «Hello, World!» будет направлена браузеру, который получит ее в том же виде, какой она имеет в листинге 22, разве что содержание тега `<TITLE>` здесь другое. Да еще мы видим, что перед собственно текстом на HTML браузеру направляется строка служебной информации о характере передаваемых данных (`text/html`). Впрочем, и в случае обычной HTML-страницы такие сведения передаются веб-сервером при пересылке страниц.

В отличие от обычной, статической HTML-страницы, которая хранится на сервере в неизменном виде, в случае, когда страница формируется CGI-программой «на лету», она при каждом просмотре может быть разной. В приведенном примере такого, конечно, нет, но представить это совсем нетрудно.

Перл

Очень часто CGI-скрипты стали писать на языке Перл (Perl). Как и в случае со скриптами на Java Script и VB Script, сказалась простота использования. Программы на Перле исполняются прямо по их исходному коду в режиме интерпретации. Поэтому их часто называют скриптами (CGI-скриптами). Предварительная компиляция не требуется. Чтобы проверить действие Перл-программы достаточно поместить файл с её текстом в соответствующий каталог сервера и вызвать веб-страницу, на которой есть ссылка на CGI-скрипт.

Вторым фактором, повлиявшим на популярность Перла при CGI-программировании, стало наличие в языке разнообразных возможностей работы с текстами. Дело в том, что в соответствии с интерфейсом CGI обмен информацией между веб-страницей и программой происходит при помощи текстовых сообщений, с которыми приходится выполнять разнообразные преобразования. В файлах на сервере данные часто также хранятся в текстовом виде. Как и в случае с JavaScript, есть масса примеров готовых скриптов, доступных в виде исходного текста на Перл. Вот и мы рассмотрим простейший пример написанного на Перле CGI-скрипта (листинг 1.26).

Листинг 1.26. CGI-скрипт «Hello, World!» на Перле

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "<HTML>\n";
print "<HEAD>";
print "<TITLE>Привет от CGI-скрипта</TITLE>";
print "</HEAD>";
print "<BODY>\n";
print "Hello, World!\n";
print "</BODY>\n";
print "</HTML>\n";
```

Существенных особенностей Перла в этом простейшем примере, конечно, не увидеть. Хотя даже здесь заметно, что язык не требует обязательных вводных и замыкающих элементов. Программа просто состоит из последовательности исполняемых команд. Это одно из правил, упрощающих написание простых скриптов и снижающих «барьер вхождения» при освоении Перла.

Первая строка, указывающая расположение Перл-интерпретатора в системе, не является командой собственно Перла. По правилам

языка — это комментарий, но он содержит директиву для исполняющей системы. Такое указание бывает необходимо.

Хорошо совместимые интерпретаторы Перла существуют для основных операционных систем, в первую очередь Unix и Windows. Это позволяет использовать одну и ту же программу без изменений или с минимальными изменениями для работы на разных веб-серверах. Напомню, что для этого не требуется даже перекомпиляция, поскольку программы на Перле интерпретируются.

Что касается самого языка Перл, то, как порождение Unix, он несет в себе в первую очередь черты языков, применяемых в этой системе. Это Си, командные языки Unix, специализированные языки для обработки текстов (awk), встроенные языки текстовых редакторов. Можно найти и черты, роднящие Перл с Бейсиком: необязательность описаний, наличие специальных конструкций там, где другие языки обходятся единообразным синтаксисом. Вряд ли можно считать, что Перл представляет собой существенный шаг в эволюции языков программирования. Синтаксис языка довольно запутан, контроль типов слаб. Устройство языка не способствует надёжному программированию, допущенные программистом ошибки могут долгое время оставаться незамеченными.

Ява-сервлеты

Сделав вначале упор на использование языка Ява для написания аплетов, компания Sun Microsystems (впоследствии Oracle, поглотившая Sun) в дальнейшем стала переносить акцент на применение Явы для программирования веб-серверов. Действительно, не существует никаких препятствий в использовании этого языка для CGI-программирования. Наоборот, в силу того, что язык Ява платформенно-независим, его применение для этих целей — разумный выбор. Независимость CGI-программы от операционной системы и веб-сервера — её несомненный плюс. Кроме того, по сравнению с Перл, Ява гораздо более «правильный» язык, лучше подходящий для создания надежных систем. Что же касается сервисных возможностей (средства обработки текста, CGI-запросов), то они легко реализуются с помощью соответствующих библиотек.

Для поддержки Ява-программ, выполняемых на сервере, был предложен специальный программный интерфейс, а сами такие программы названы *сервлетами*. Как разъясняет компания Sun, сервлет можно рассматривать как аплет, исполняемый на сервере.

Хотя создатели технологии сервлетов отделяют их от CGI-программ, указывая на ряд отличий, разница невелика. Это можно видеть на примере сервлета (листинг 1.27), формирующего страницу «Hello, World!».

Листинг 1.27. Сервлет «Hello, World!»

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {
   protected void doGet(HttpServletRequest req,
                        HttpServletResponse res
      ) throws ServletException, IOException
   {
      PrintWriter out = res.getWriter();
      res.setContentType("text/html");
      out.println("<HTML>");
      out.println("<HEAD>");
      out.println("<TITLE>"+
         Привет от сервлета \"Hello, World!\"+
         "</TITLE>");
      out.println("</HEAD>");
      out.println("<BODY>");
      out.println("Hello, World!");
      out.println("</BODY>");
      out.println("</HTML>");
      out.close();
   }
}
```

Принцип устройства CGI-программ и здесь соблюден. Только обставлено это «стандартными заклинаниями» в характерном для Явы многословном стиле. Писать такой текст, конечно, сложнее, чем аналогичную программу на Перле, и «барьер вхождения» будет выше. Расчет на то, что при разработке больших систем надежность Явы перевесит.

Языки активных серверных страниц

Используя CGI-программы при создании интернет-сайтов, приходится формировать одну веб-страницу из нескольких компонентов. В одном файле находится HTML-код страницы, в других — CGI-скрипты. Возможно, однако, объединение HTML-кода и кода выпол-

няемых на сервере программ в одном файле. Это объединение предлагают технологии PHP, ASP, JSP.

Принцип устройства и работы таких «активных серверных страниц» следующий. В HTML-страницу записываются специальные теги, содержащие текст программ-скриптов, которые должны выполниться сервером при передаче страницы в браузер. Страница размещается на веб-сервере. Сервер при запросе страницы фильтрует ее содержимое, направляя часть, содержащую собственно HTML-код, по сети браузеру, а программы, записанные в специальных тегах, выполняя в режиме интерпретации. Чтобы иметь возможность распознавать специальные теги с программами-скриптами и исполнять их, сервер должен быть оснащен соответствующей поддержкой.

Описанный способ программирования реализован в технологии PHP (Personal Home Page tools — инструмент для персональных страниц), которая повсеместно применяется на Unix-серверах и поддерживается некоторыми веб-серверами, работающими под управлением Windows. Помещаемые в текст страницы *программы-скрипты записываются на языке, напоминающем Си и Перл и также называемом PHP*. Вот пример использования PHP для создания страницы «Hello, World!» (листинг 1.28).

Листинг 1.28. «Hello, World!» на PHP

```
<HTML>
   <HEAD>
      <TITLE>Привет от PHP</TITLE>
   </HEAD>
   <?PHP
      echo "<BODY>
      Hello, World!
      </BODY>"
   ?>
</HTML>
```

Как можно догадаться, при просмотре этой страницы в браузере мы увидим то же, что и в предыдущих примерах. Важно при этом понимать, что браузер получит не приведенный в листинге 1.28 текст, а «чистый» HTML-код, такой же, как в листинге 1.22. Тег `<?PHP … ?>` будет обработан на сервере, PHP-команда `echo` будет исполнена, в результате чего в HTML-код попадет то, что она «печатает». В этом примере она печатает не только само сообщение, но также теги `<BODY>` и `</BODY>`.

Технология, аналогичная PHP, созданная компанией Microsoft и ориентированная на Windows-серверы, называется ASP (Active Server Pages — активные серверные страницы). *Скрипты для ASP пишут на диалекте Visual Basic*.

Предложенная Sun Microsystems технология активных серверных страниц, в которой *для написания скриптов используется Ява-подобный язык*, получила название *JavaServer Pages* (JSP), а программы-скрипты именуют *скриптлетами*. В листинге 1.29 приведен пример JSP-скриптлета.

Листинг 1.29. Скриптлет «Hello, World!»

```
<HTML>
   <HEAD>
      <TITLE>Привет от JavaServer Page</TITLE>
   </HEAD>
   <BODY>
      <% out.println("Hello, World!"); %>
   </BODY>
</HTML>
```

Обрабатывая скриптлет, Java Web Server преобразует его в сервлет — программу на языке Ява, которая и выполняется на сервере. Как нетрудно видеть (сравните листинги 1.27 и 1.29), скриптлеты, использующие лишь элементы Явы, избавлены от обязанности соблюдать все формальности языка, и оттого оказываются проще.

Языки Интернета: повторение пройденного

Суммируя сказанное по поводу языков Интернета, приведу таблицу (табл. 1.2), систематизирующую основные схемы применения программ и языков программирования для создания веб-страниц.

Таблица 1.2. Технологии и языки веб-программирования

	Исполнение на сервере	**Исполнение у клиента**
Программа размещается внутри HTML-кода	Активные серверные страницы: PHP, ASP, JSP	Скрипты: Java Script, VB Script
Программа размещается отдельно	CGI и сервлеты: Перл, Си, Паскаль, Ява и др.	Аплеты: Ява

Ситуацию с языками, получившими распространение в период бурного развития сети Интернет, можно сравнить с тем, что происходило в конце 1970-х — начале 1980-х годов, во времена, когда появились персональные компьютеры (ПК). Общность заключается в том, что и тогда и теперь круг тех, кто пишет программы, быстро и существенно расширился. С появлением ПК возможность программировать получили научные работники и инженеры, преподаватели и студенты, и не в последнюю очередь любознательные молодые люди школьного возраста.

Основным языком для программирующих непрограммистов стал Бейсик. Оно и понятно. Научиться писать небольшие программы на Бейсике можно было буквально за несколько часов. При малом размере программ, написанных «для себя», недостатки Бейсика не играли особой роли. Не имея опыта разработки больших программных систем и систематической программистской подготовки, программисты-любители с трудом могли понять необходимость соблюдения технологической дисциплины и строгих правил таких языков, как, например, Паскаль. Все это способствовало широкому распространению Бейсика, который пережил тогда свое второе рождение, в то время как более совершенные языки оказались на втором плане.

В дальнейшем ситуация нормализовалась. Большинство потребностей бывших программистов-любителей стали удовлетворяться программами промышленной разработки, и необходимость в программировании для многих людей исчезла. Для вычислений, в том числе и довольно сложных, стали применять электронные таблицы и мощные математические пакеты, а, например, разработка компьютерных игр превратилась в отдельную отрасль. В роли инструментов при создании программ массового применения использовались в основном традиционные языки программирования, созданные до появления ПК (Си, Паскаль и др.). Потребности, возникшие при создании графических пользовательских интерфейсов, стимулировали появление и распространение объектно-ориентированных языков и диалектов (Си++, Объектный Паскаль). Наконец, свидетельством зрелости отрасли программирования стало создание и продвижение в конце 1990-х — начале 2000-х годов новых языков Ява и Си#, которые, несмотря на ряд недостатков, утверждают фундаментальные ценности: систематичность, строгость, надежность.

С массовым распространением Интернета вновь расширился круг тех, кто пишет программы «для себя». Желание украсить динамическими элементами веб-страницы возникает у многих. А самый удобный язык в этой ситуации тот, на котором можно быстро написать несложную программу. Здесь и возникают Перл, Java Script, PHP — языки, обладающие минимальным барьером вхождения и не требующие от программиста соблюдения чрезмерных формальностей, вроде обязательного описания переменных, типов и т. п. Замечено, что изучение таких языков, как правило, строится исключительно на примерах. И начинающий интернет-программист обычно просто модифицирует для своих целей готовую программу.

По мере «взросления» Интернета роль самодеятельного программирования уменьшилась, а типовые потребности, им удовлетворяемые, покрываются инструментами, не требующими программирования вообще. В то же время возросла роль универсальных, надежных языков профессионального программирования, таких как Ява и Си#.

Какой язык лучше. Сравнительная оценка языков программирования

По-английски я говорю с торговцами, по-итальянски с женщинами, по-французски с мужчинами, по-испански с Богом, а по-немецки с моей лошадью.

Приписывается испанскому королю Карлу V

Достаточно часто можно услышать вопрос о том, какой язык программирования предпочтительней. Мне его обычно задают студенты, научившиеся программировать на Паскале и выбирающие для изучения второй язык.

Универсального ответа на вопрос о лучшем языке, конечно, нет. Можно говорить лишь о выборе языка для конкретной задачи, а точнее, для конкретных обстоятельств. Многое определяется не только свойствами собственно языка, но наличием, качеством и доступностью соответствующей системы программирования, библиотек программ (модулей, компонентов, классов), учебной и справочной литературы по языку и системе программирования, возможностью получать консультации. Наконец, следует принять во внимание личные предпочтения программиста: язык, которым специалист лучше владеет, возможно, и будет лучшим.

Выбор языка редко бывает свободным. Разработка крупных программных комплексов ведется большими коллективами специалистов на протяжении длительного времени. Программист, участвующий в таком проекте, скорее всего, вообще не обладает свободой в выборе языка программирования. Но и при индивидуальной работе свобода выбора ограничивается массой обстоятельств, некоторые из которых были названы выше.

Языки программирования имеют формальную природу. Их правила формулируются точно и строго. Отсюда, однако, не следует, что сравнительная оценка языков также может быть однозначна. Формальные языки применяются в отнюдь не формальной среде. Оценить, тем более количественно, как то или иное свойство языка и язык в целом влияют на успешность его применения, способствуют (или препятствуют) снижению затрат на разработку и эксплуатацию программы совсем не просто. Источником информации, дающей основу для подобных сравнений, могли бы быть экспериментальные данные. Но достаточно сложно представить масштабный эксперимент, в котором разработка реальной, а значит большой и сложной системы, велась бы одновременно на нескольких языках. В связи с этим в основе сравнения языков программирования лежат, как правило, экспертные оценки.

С момента появления языков высокого уровня их распространение задевало экономические интересы крупных компаний. Фортран и ПЛ/1 — это языки IBM, Visual Basic и Си# — детища Microsoft, Ява — Sun. Компании-разработчики всегда рекламировали свои разработки, а реклама, как мы хорошо знаем, преувеличивает достоинства товара и совсем не упоминает о его недостатках. Ярким примером стало продвижение фирмой Sun языка Ява посредством беспрецедентной по размаху рекламной кампании. Реклама действенна, поэтому в общественном сознании формируется зачастую искаженное представление не только о выгоде от возможного применения того или иного языка, но и о его собственных характеристиках.

В этих условиях важную роль играют систематизированные данные о языках программирования, которые могут служить для их обоснованного выбора. Весь предыдущий обзор можно рассматривать как попытку такой систематизации. Ниже приводятся экспертные оценки языков программирования и количественные данные, полученные при исследовании их отдельных свойств.

Арифметика синтаксиса

Синтаксис языка программирования определяет внешнюю форму программы, правила ее записи, не касаясь смысла языковых конструкций, их семантики. В то же время синтаксические единицы языка, такие как, например, модуль, класс, блок, оператор, выражение, являются одновременно и его основными смысловыми понятиями. Система понятий языка решающим образом определяет его восприятие, в том числе лёгкость понимания, освоения и использования.

Начиная с Алгола-60, описания синтаксиса (грамматики) языков программирования строго формализованы. Это позволяет определить объективные количественные характеристики таких описаний для различных языков и провести их сопоставление. Нужно только представить синтаксические правила различных языков в едином формате. В качестве такого формата была выбрана РБНФ — Расширенная Бэкуса — Наура Форма — вариант БНФ, использованный Н. Виртом при описании синтаксиса языка Оберон. Были исследованы языки Алгол-60, Паскаль (авторский вариант Н. Вирта), Си (K&R, стандарт ANSI 1989 года и стандарт ISO/IEC 1999 года), Си++ (авторский вариант Б. Строуструпа 1990 года и стандарт ISO/IEC 1998 года), Модула-2, Ада, Турбо-Паскаль (версии 2.0, 5.0, 5.5, 6.0), Объектный Паскаль (входной язык системы Delphi версий 1.0 и 7.0), Оберон, Оберон-2, Ява, Ява-2, Си#.

Критерии сравнения

Правила, записанные на РБНФ (как и текст на языке программирования), состоят из отдельных элементов — лексем. Лексемами являются названия понятий, именуемые в теории формальных языков нетерминальными символами или просто нетерминалами. Например, в правиле

```
ПоследовательностьОператоров = Оператор {";" Оператор}.
```

нетерминалами являются `ПоследовательностьОператоров` и `Оператор`. Терминальные символы — это те знаки, из которых и состоит в конечном счете программа (terminal — конечный, заключительный). При записи на РБНФ терминальные символы записываются в кавычках. В приведенном примере один терминал – `";"`. Терминальные символы — это тоже лексемы РБНФ. Наконец, к числу лексем относятся специальные знаки, используемые в самой РБНФ. В правиле

про последовательность операторов это знак равенства, фигурные скобки и точка в конце. Всего в приведенном правиле 8 лексем.

Общее количество лексем в описании синтаксиса языка может служить обобщенной характеристикой размера этого описания. Назовем эту величину **объемом синтаксиса**. Число лексем использовать в качестве меры объема гораздо лучше, чем, скажем, число знаков в описании. В этом случае значение критерия не будет зависеть от того, на каком языке (русском, английском) или какими конкретно словами названы нетерминалы — понятия языка.

Количество различных нетерминалов — следующая характеристика, которую мы будем вычислять. Количество используемых для описания языка понятий — важнейшее свойство, от которого зависит легкость освоения этого языка. Можно заметить, что число нетерминалов должно быть равно **числу правил** в описании синтаксиса, поскольку для каждого понятия обязано существовать одно правило.

Набор и **количество различных терминальных символов** языка, упомянутых в синтаксических формулах, характеризуют лексику языка — набор знаков и специальных символов.

Во всех обсуждаемых нами языках существуют служебные слова, которые могут употребляться только в строго определенном смысле. Их программист, вообще-то, должен знать наизусть. Поэтому небезынтересен будет подсчет **количества служебных слов**.

Названные характеристики могут служить мерой сложности языка (точнее, его грамматики).

Результаты измерений

В таблице 1.3 представлены результаты обработки описаний синтаксиса. Подсчет выполнен с помощью специальной программы, которая устроена на тех же принципах, что и компиляторы языков программирования. Ведь РБНФ-нотация сама представляет собой язык со своей лексикой и синтаксисом. Программа проверяет корректность описаний, в ходе анализа строятся таблицы и ведутся подсчеты.

Таблица 1.3. Характеристики синтаксиса языков программирования

Язык	Объем синтаксиса (лексем)	Количество правил (непустых)	Количество термин. символов	Количество служебных слов
Алгол-60	890	102 (90)	88	24
Паскаль	1004	109 (85)	84	35

Продолжение табл. 1.3

Язык	Объем синтаксиса (лексем)	Количество правил (непустых)	Количество термин. символов	Количество служебных слов
Модула-2	887	70 (69)	88	39
Оберон	765	62 (59)	90	32
Оберон-2	726	43 (42)	91	34
Ада	2167	221 (166)	100	63
Ада 95	2999	327 (258)	98	69
Turbo Pascal 2.0	1176	123 (99)	87	42
Turbo Pascal 5.0	1325	125 (105)	87	48
Turbo Pascal 5.5	1402	133 (112)	87	52
Turbo Pascal 6.0	1479	141 (117)	89	55
Объектный Паскаль (Delphi 1.0)	1768	171 (138)	90	83
Объектный Паскаль (Delphi 7.0)	2041	186 (165)	92	107
Си (K&R)	913	52 (49)	122	27
ANSI Си (ISO/IEC 9899:1990)	1109	83 (80)	125	32
ANSI Си с языком препроцессора	1223	90 (87)	129	42
Си 99 (ISO/IEC 9899:1999)	1413	110 (106)	133	37
Си 99 с языком препроцессора	1801	134 (126)	145	47
Си++ (Строуструп, 1990)	1654	124 (117)	131	48
Си++ (ISO/IEC 14882-1998)	2292	176 (166)	136	63
ISO/IEC Си++ с языком препроцессора	2667	197 (184)	148	84

Продолжение табл. 1.3

Язык	Объем синтаксиса (лексем)	Количество правил (непустых)	Количество термин. символов	Количество служебных слов
Ява	1813	172 (158)	121	48
Ява-2	2055	193 (179)	121	49
Си#	3036	295 (268)	115	88
Си# с языком препроцессора	3768	346 (313)	135	97

ЛАНЬ®

В графе «Количество правил» в скобках указано число «непустых» правил, которые вводятся в синтаксис для облегчения его восприятия. Например, в грамматике Ады есть правило:

```
ОпределениеЦелогоТипа = ОграничениеДиапазона.
```

Его легко можно было бы исключить, заменив повсеместно в других правилах понятие `ОпределениеЦелогоТипа` на `ОграничениеДиапазона`. Но это не сделано, чтобы придать грамматике более содержательную форму.

Обязательным компонентом любой системы программирования на Си, Си++ и Си# является препроцессор. Хотя его директивы — это не часть собственно языка, но без них не обходятся реальные программы, и знание этих директив программисту необходимо. В первом издании знаменитой книги Б. Кернигана и Д. Ритчи, определяющей авторский диалект Си, нет формальной спецификации языка препроцессора, в документах же по ANSI Си, ISO/IEC Си++ и Си# она имеется. В таблице приводятся метрики синтаксиса этих языков, как с включением, так и без включения языка препроцессора.

ЛАНЬ®

На рисунке 1.21 приведена диаграмма, которая позволяет наглядно увидеть различия в объеме синтаксиса языков, появившихся в разное время.

Выделяются несколько «линий языков». Во-первых, «линия Вирта» — языки, созданные Никлаусом Виртом: Паскаль, Модула-2, Оберон, Оберон-2. Для этой группы характерно упрощение синтаксиса по мере усовершенствования. Это — принципиальная позиция Н. Вирта — увеличение мощности языка без роста и даже с уменьшением его сложности. Нет сомнений, что Модула-2 и Оберон обладают большей выразительной силой по сравнению со стандартным Паскалем, но оказываются заметно проще его. Может вызывать недоумение меньший по сранению с Обероном объем синтаксиса Оберона-2.

Ведь Оберон-2 — почти точное надмножество Оберона. Дело здесь в том, что в спецификации Оберона-2 сделаны редакционные изменения, уменьшившие число правил. Например, отдельные правила для различных видов операторов объединены в одно, определяющее понятие `Оператор`.

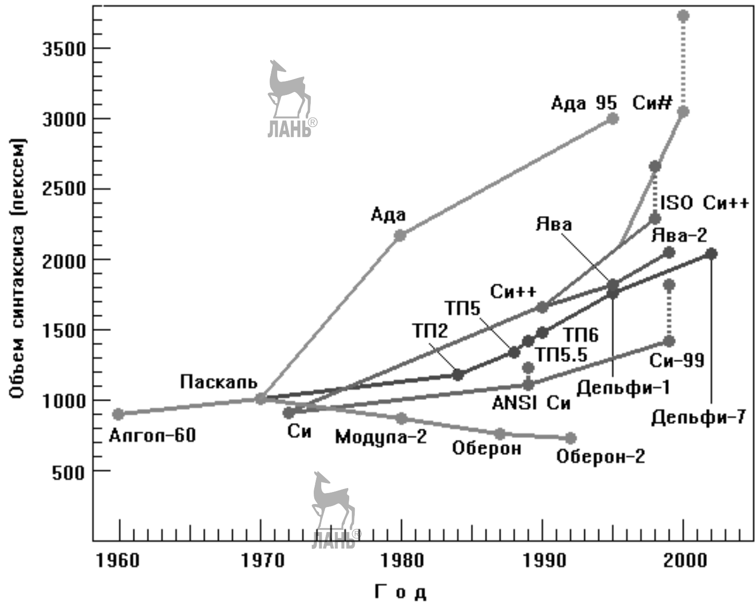


Рис. 1.21. Объем синтаксиса языков программирования

Другие направления развития языков, как можно видеть, связаны с постоянным ростом сложности их синтаксиса. Рост сложности вдоль «линии Borland» (Паскаль – Turbo Pascal – Объектный Паскаль) легко объясним. Входной язык каждой новой версии систем программирования Borland, за малым исключением, является расширением языка предыдущих версий. Тем самым обеспечивается работа старых программ в новых системах и плавный переход прикладных программистов к использованию новых версий.

Обращает на себя внимание существенно больший объем синтаксиса языка Си# по сравнению с языками Си++ и Ява, тем более — по

сравнению с Си, в то время как сложность синтаксиса Си++ (ISO/IEC) и Ява сопоставимы. Сравнение этих языков по числу лексем представляется в максимальной степени корректным, поскольку их грамматики, использованные в исследовании, записаны в одной и той же исходной нотации, более того, обладают очевидной преемственностью. В них много общих терминов, есть одинаковые правила. Означают ли приведенные данные, что, например, Си# заметно сложнее Си++, а Ява лишь ненамного проще Си++? И да, и нет.

Различия в объеме синтаксиса напрямую влияют на сложность синтаксических анализаторов, являющихся частью компиляторов. Впрочем, синтаксический анализатор, хоть и очень важная, но не самая сложная часть программы-компилятора. Гораздо больше усилий требуется от разработчиков для создания подсистем компилятора, ответственных за семантическую составляющую языка. С другой стороны, каждое правило грамматики порождает и необходимость в определенной семантической обработке. Таким образом, «с позиции компьютера» объем синтаксиса напрямую влияет на сложность синтаксического анализатора, а косвенно, причем не обязательно в прямой пропорции, — на весь компилятор.

А как обстоит дело с субъективным восприятием языка? Можно, по-видимому, сказать, что *при прочих равных условиях* объем синтаксиса позволяет судить и о сложности языка с точки зрения программиста. Но, пожалуй, в первую очередь не о сложности понимания, а о сложности, относящейся к «размеру» языка. При этом больший размер не обязательно будет осложнять применение — многие языки обладают тем свойством, что их можно использовать, не зная языка целиком. Не надо забывать, что в «большом» языке обычно предусмотрено и больше возможностей.

Равенство упомянутых выше *прочих условий* можно наблюдать в паре Ява – Си#. Эти языки имеют одинаковую объектную модель, сходство семантики, общего предка. И можно, основываясь на результатах выполненных измерений, твердо заявить, что Си# существенно *больше* Явы. Рассматривая это различие по существу, вспомним, что в Си# есть ряд элементов, отсутствующих в Яве: структуры (`struct`), перечисления (`enum`), свойства, индексаторы, небезопасный код. При этом, ничего не зная, например, про блоки небезопасного кода, можно вполне успешно писать программы на Си#.

Что касается сложности понимания, восприятия языка программистом как сложного или наоборот простого, то приведенные цифры позволяют сделать некоторые суждения на этот счет (обратите внимание на измеренное количество грамматических *понятий* — правил языка на рисунке 1.22), но эти суждения отнюдь не однозначны.

Мы только что видели это на примере Си# и Явы. Несмотря на существенные отличия в объеме синтаксиса, язык Си# не должен, вероятно, рассматриваться как принципиально более сложный в сравнении с Явой.

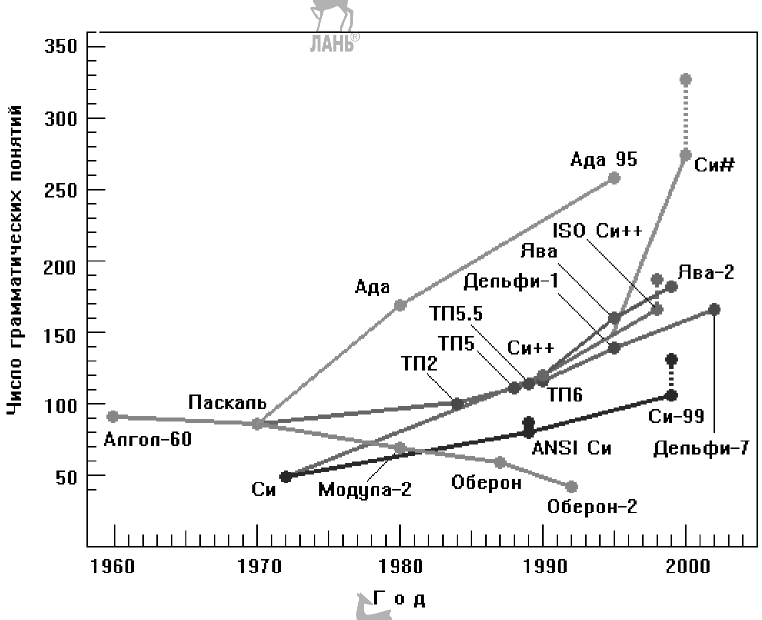


Рис. 1.22. Количество грамматических понятий (нетерминалов, правил грамматики) языков программирования

В то же время можно твердо заявить, что Оберон намного проще Явы и Си# (впрочем, это было понятно и без цифр). Наблюдая огромную разницу в объемах синтаксиса и числе понятий, зная, что возможности этих языков вполне сопоставимы, приходим к выводу, что Оберон имеет радикальное преимущество.

Вернемся к вопросу о сопоставлении Явы и Си# с языком Си++. Объем синтаксиса Си# намного больше, чем у Явы и Си++. Вместе с тем, по мнению многих экспертов, Си# и Ява воспринимаются как языки более простые в сравнении с Си++. Если так, то Си++ — плохо организованный язык. Имея *относительно* простой формальный синтаксис, он воспринимается как язык сложный, поскольку этот синтаксис неудачен и неадекватен сопоставляемой ему семантике.

Важнейшие языки

И все же, ответ на вопрос о самых главных языках есть. Первый, наиважнейший язык, которым хорошо, а лучше, если в совершенстве, должен владеть каждый программист — это его родной язык. Действительно, разве можно себе представить, что, не умея сформулировать задачу на русском, не выразив обычными словами способы ее решения, можно спроектировать и реализовать программную систему? Мышление неотделимо от языка, и естественный язык при решении технических задач не менее (а скорее, более) важен, чем язык программирования.

Программирование во многом сродни литературному творчеству. Суть и того и другого занятия — сочинительство, придумывание нового. Из простых известных слов складываются новые проза и стихи, и программы.

Ну а второй (для кого-то он — первый) важный язык — английский. Так сложилось, что центром компьютерного мира стала англоговорящая Америка. Спецификации многих языков, масса другой документации существуют только на английском. Европейский вклад в развитие языков программирования также очень велик. В создании Алгола-60 участвовало много европейских специалистов, сам Алгол символизирует европейскую языковую традицию, а все самые распространенные современные языки, по сути, алголоподобны. Тот же Н. Вирт, родившийся и живущий в немецко-говорящей части Швейцарии, формулирует описания своих языков по-английски.

Глава 2. Теоретические основы трансляции

Языки программирования — искусственно созданные формальные системы, которые могут изучаться математическими методами. Теория формальных языков и грамматик — это обширная область математики, примыкающая к алгебре, математической логике и теории автоматов. Цель этой главы — познакомиться с понятиями и результатами теории формальных языков и грамматик в той мере, как это требуется для понимания принципов конструирования трансляторов.

Формальные языки и грамматики

Основные термины и определения

Введем в обиход основные понятия, которые будут использованы в определении формального языка.

Алфавит — *конечное непустое множество символов.*

Термин *символ* следует понимать здесь в самом широком смысле. Это может быть буква, цифра или знак препинания. Но символом можно считать и любой другой знак, рассматриваемый как нечто неделимое — служебное слово языка программирования, иероглиф и т. д. Будем обозначать алфавиты буквой Σ (сигма).

Примеры алфавитов:

$\Sigma_1 = \{0, 1\}$;

$\Sigma_2 = \{a, b, c\}$;

$\Sigma_3 = \{A, B, C, \ldots, Z, a, b, c, \ldots, z,$ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, –, *, ..., **`div`**, ..., **`program`** $\}$

Алфавит — это множество, поэтому при перечислении его элементов использованы фигурные скобки, как это принято в математике. Алфавит Σ_1 содержит два символа, алфавит Σ_2 — три. Под Σ_3 подразумевается алфавит языка Паскаль.

Цепочка *над алфавитом* Σ — *произвольная конечная последовательность символов из* Σ.

Примеры цепочек над алфавитом Σ_2:

$\alpha = abbca$;

$\beta = ab$;

$\gamma = ba$;

$\delta = c$.

Цепочки будем обозначать греческими буквами.

Пустая цепочка — *цепочка, не содержащая символов* (*содержащая ноль символов*). *Обозначается буквой* ε.

Если α и β — цепочки, то запись $\alpha\beta$ означает их *конкатенацию* (склеивание), то есть $\alpha\beta$ — это цепочка, образованная приписыванием к цепочке α цепочки β справа.

Если α — цепочка, то α^n означает цепочку, образованную n-кратным повторением цепочки α:

$$\alpha^n = \underbrace{\alpha\alpha\ldots\alpha\alpha}_{n \text{ раз}}.$$

В частном случае, если a — символ, то $a^n = \underbrace{aa\ldots aa}_{n \text{ раз}}$.

Будем обозначать Σ^* — (бесконечное) множество всех цепочек над алфавитом Σ, включая пустую цепочку; Σ^+ — множество всех цепочек над алфавитом Σ, не включая пустой цепочки. Например, если $\Sigma_1 = \{0, 1\}$, то $\Sigma_1{}^*$ представляет собой множество всех цепочек, которые могут быть составлены из символов 0 и 1. В это множество входят пустая цепочка, все цепочки, состоящие из одного символа, все цепочки, состоящие из двух символов и т. д.: $\Sigma_1{}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$.

$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$, где $\cup$ — знак операции объединения множеств.

Теперь можно дать и определение формального языка.

Языком *над алфавитом* Σ *называется произвольное множество цепочек, составленных из символов* Σ.

Будем обозначать язык над алфавитом (с алфавитом) Σ — $L(\Sigma)$ или просто L, если алфавит ясен из контекста.

Таким образом, речь идет о том, что язык — это некоторое, тем или иным образом определенное, подмножество множества всех цепочек, которые могут быть построены из символов данного алфавита. $L(\Sigma) \subseteq \Sigma^*$. На рисунке 2.1 наглядно показано соотношение множеств Σ^* и $L(\Sigma)$.

Принадлежащие языку цепочки называют также *предложениями* языка.

Еще раз отметим, что множество цепочек Σ^* всегда бесконечно, в то время как множество цепочек, образующих язык, может быть и конечным. Практический интерес представляют, конечно, языки, содержащие бесконечное множество цепочек; к числу таких языков относятся и языки программирования.

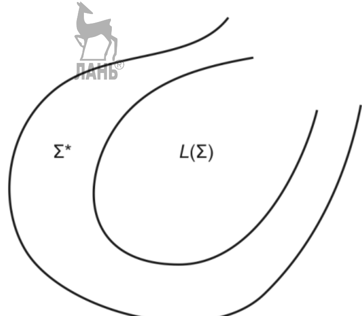


Рис. 2.1. Язык — подмножество множества всех цепочек над алфавитом Σ

Примеры языков

Пример 1. Определим язык $L_1 = \{a^n b^n \mid n \geq 0\}$, используя принятую в теории множеств нотацию, как множество всех цепочек, содержащих вначале некоторое количество символов a, а затем такое же количество символов b. Заметим, что L_1 включает и пустую цепочку, поскольку n может равняться нулю.

Записанное выше правило, определяющее язык L_1, разделяет все цепочки над алфавитом $\{a, b\}$, то есть состоящие из символов a и b, на принадлежащие L_1 и не принадлежащие ему.

Примеры цепочек, принадлежащих языку:

$\varepsilon \in L_1$ — пустая цепочка принадлежит L_1;

$ab \in L_1$ — цепочка из одной буквы a, за которой следует b;

$aaabbb \in L_1$.

Цепочки, не принадлежащие языку L_1:

$aaab \notin L_1$ — неодинаковое количество символов a и b;

$abba \notin L_1$ — порядок следования символов не соответствует определению L_1.

Пример 2. Язык $L_2 = \{a^n b^n c^n \mid n \geq 0\}$ — множество всех цепочек, содержащих вначале некоторое (возможно нулевое) количество симво-

лов a, затем такое же количество символов b, затем — столько же букв c. Например, $aaabbbccc \in L_2$, в то время как $aaabbc \notin L_2$.

Далеко не всегда удается определить язык, особенно если речь идет о языках, представляющих практический интерес, используя нотацию, примененную при определении L_1 и L_2. Значительная часть последующего материала будет посвящена рассмотрению порождающих грамматик, позволяющих компактно и однозначно определить обширный класс формальных языков. Пока же, в следующих примерах, дадим словесное описание некоторых представляющих интерес языков.

Пример 3. Рассмотрим язык правильных скобочных выражений, составленных только из круглых скобок, известный также как *язык Дика*. Обозначим его L_3. Алфавит языка Дика — это множество из двух символов — открывающей «(» и закрывающей «)» скобок: $\Sigma_3 = \{ (,) \}$. Цепочки, содержащие правильно расставленные скобки принадлежат языку Дика, все остальные последовательности открывающих и закрывающих круглых скобок — нет. Например: $(())()() \in L_3$; $()(()))(\notin L_3$.

Пример 4. Язык L_4 — множество всех цепочек, содержащих одинаковое количество символов a и b. Несмотря на простое «устройство», задать язык L_4 формулой, подобной формулам для L_1 или L_2, оказывается затруднительно. Можно заметить, что рассмотренный ранее язык L_1 является подмножеством языка L_4: $L_1 \subseteq L_4$, поскольку любая цепочка, принадлежащая L_1, принадлежит и языку L_4. Но не наоборот. Так,

$$aabb \in L_1,\ aabb \in L_4,\ abba \in L_4,\ \text{но}\ abba \notin L_1.$$

Пример 5. В качестве языка L_5 рассмотрим множество всех правильных арифметических выражений языка Паскаль, составленных из символов алфавита $\Sigma_5 = \{a, b, c, +, -, *, /, (,) \}$. Например, $a*(b+c) \in L_5$, но $c{+}{+} \notin L_5$.

Порождающие грамматики (грамматики Н. Хомского[48])

Порождающие грамматики — это простой и мощный механизм, позволяющий задавать обширный класс языков, содержащих беско-

---

[48] Ноам Хомский (Noam Chomsky, р. 1928) — американский лингвист и политический активист. Начиная с 1957 года, опубликовал ряд работ, заложивших основы математической лингвистики. Выступал против войны США во Вьетнаме, автор нескольких десятков книг на политические темы. Участник движения антиглобалистов. В отечественной прессе его фамилия иногда звучит как Чомски, однако в научных публикациях на русском языке, начиная с 1962 года, принято написание Хомский.

нечное множество цепочек. С помощью порождающих грамматик мы сможем, в частности, определить языки L_3, L_4 и L_5, для задания которых раньше ограничивались словесными формулировками. Порождающие грамматики используются и при описании синтаксиса языков программирования.

Порождающей грамматикой называется четверка:

$$G = (T, N, P, S),$$

где T — конечное *множество терминальных* (*основных*) *символов* — основной алфавит. Элементами множества T являются символы, из которых в конечном итоге и состоят цепочки языка, порождаемого данной грамматикой. Терминальный (от *лат.* terminus — предел, конец) и означает «конечный, концевой». T — это не что иное, как алфавит языка, порождаемого грамматикой. В дальнейшем, если не оговорено особо, терминальные символы или просто *терминалы* будут обозначаться малыми буквами латинского алфавита: *a, b, c* и т. д.

N — конечное *множество нетерминальных* (*вспомогательных*) *символов* — вспомогательный алфавит. Нетерминальные символы, по-другому *нетерминалы,* — это понятия грамматики (языка), которые используются при его описании. Нетерминалы будем обозначать заглавными латинскими буквами: *A, B, C, D, E* и т. д.

P — конечное *множество правил вывода*, называемых также *продукциями*. Каждое правило множества P имеет вид:

$$\alpha \to \beta,$$

где α и β — цепочки терминальных и нетерминальных символов. Цепочка α не должна быть пустой, цепочка β может быть пуста: $\alpha \in (T \cup N)^+$; $\beta \in (T \cup N)^*$. Правило $\alpha \to \beta$ определяет возможность *подстановки* β вместо α в процессе вывода (порождения) цепочек языка.

S ($S \in N$) — *начальный символ грамматики* — один из множества нетерминальных символов, *начальный (стартовый) нетерминал*. Начальный нетерминал — это понятие, соответствующее правильному предложению языка. Например, начальный нетерминал грамматики выражений обозначает «выражение», а начальный нетерминал грамматики языка Паскаль — «Программа».

Примеры грамматик. Порождение предложений языка

Пример 1. Рассмотрим грамматику

$$G_1 = (\{a, b\}, \{S\}, \{S \to aSb, S \to \varepsilon\}, S).$$

Здесь все элементы четверки записаны явно. Множество терминальных символов $T = \{a, b\}$; множество нетерминалов содержит один элемент: $N = \{S\}$, а множество правил — два: $P=\{S \to aSb, S \to \varepsilon\}$; роль начального нетерминала исполняет S.

Грамматика может использоваться для порождения (вывода) цепочек — предложений языка. Процесс порождения начинается с начального нетерминала, в нашем примере это S. Если среди правил есть такое, в левой части которого записана цепочка S, то начальный нетерминал может быть заменен правой частью любого из таких правил. Оба правила грамматики G_1 содержат в левой части S. Применим подстановку, заданную первым правилом, заменив S на aSb:

$$S \underset{(1)}{\Rightarrow} aSb.$$

К получившейся цепочке aSb снова, если удастся, можно применить одно из правил грамматики. Если в цепочке есть подцепочка, совпадающая с левой частью хотя бы одного из правил, то эту подцепочку можно заменить правой частью любого из таких правил. В цепочке aSb есть подцепочка S, совпадающая с левой частью обоих правил грамматики G_1. Мы вправе применить любое из этих правил. Используем снова правило (1) для продолжения вывода:

$$S \underset{(1)}{\Rightarrow} aSb \underset{(1)}{\Rightarrow} aaSbb.$$

Теперь к получившейся цепочке применим правило (2) ($S \to \varepsilon$), заменив S пустой цепочкой. Получим такую последовательность подстановок (саму букву ε в последней цепочке записывать, конечно, не нужно):

$$S \underset{(1)}{\Rightarrow} aSb \underset{(1)}{\Rightarrow} aaSbb \underset{(2)}{\Rightarrow} aabb.$$

Очевидно, что к получившейся цепочке ни одно из правил грамматики G_1 больше не применимо. Процесс порождения завершен. Нетрудно заметить, что с помощью грамматики G_1 можно породить любую цепочку языка $L_1 = \{a^n b^n \mid n \ge 0\}$, применив к начальному нетерминалу правило (1) ($S \to aSb$) n раз, а затем один раз правило (2). В то же время, грамматика G_1 не порождает ни одной цепочки терминальных символов, не принадлежащей языку L_1. То есть множество терминальных цепочек, порождаемых грамматикой G_1, совпадает с языком L_1. Другими словами, грамматика G_1 порождает язык L_1:

$$L(G_1) = L_1 .$$

Обычно при записи грамматики не выписывают четверку ее элементов явно. При соблюдении соглашений об обозначениях терминалов и нетерминалов достаточно записать только правила. Правила с одинаковой левой частью можно объединять в одно, отделяя альтернативные правые части вертикальной чертой. Первым записывается правило, содержащее в левой части начальный нетерминал. С учетом этого грамматика G_1 может быть записана так:

$$G_1:\ S \to aSb \mid \varepsilon .$$

Пример 2. Рассмотрим грамматику G_2 (цифры справа — номера правил).

$$
\begin{aligned}
G_2:\ & S \to aSBc && (1)\\
& S \to abc && (2)\\
& cB \to Bc && (3)\\
& bB \to bb && (4)\\
& S \to \varepsilon && (5)
\end{aligned}
$$

Проведем вывод цепочек из начального нетерминала грамматики G_2. Под стрелкой, обозначающей подстановку, будем указывать, как и раньше, номер примененного правила. Итак:

$$S \underset{(1)}{\Rightarrow} aSBc \underset{(2)}{\Rightarrow} aabcBc \underset{(3)}{\Rightarrow} aabBcc \underset{(4)}{\Rightarrow} aabbcc.$$

Еще одна серия подстановок:

$$S \underset{(1)}{\Rightarrow} aSBc \underset{(1)}{\Rightarrow} aaSBcBc \underset{(2)}{\Rightarrow} aaabcBcBc \underset{(3)}{\Rightarrow} aaabBccBc \underset{(3)}{\Rightarrow} aaabBcBcc \underset{(3)}{\Rightarrow}$$

$$aaabBBccc \underset{(4)}{\Rightarrow} aaabbBccc \underset{(4)}{\Rightarrow} aaabbbccc.$$

Можно убедиться, что грамматика G_2 порождает цепочки терминалов вида $a^n b^n c^n$ и никакие другие. Количество повторений символов в результирующей цепочке определяется тем, на каком шаге применяется правило (2). Наличие правила (5) позволяет получить пустую цепочку, если применить это правило первым, в то время как попытка использования этого правила на последующих шагах не позволит вывести цепочку терминалов. Грамматика G_2 порождает множество терминальных цепочек, совпадающее с языком $L_2 = \{a^n b^n c^n \mid n \ge 0\}$:

$$L(G_2) = L_2 .$$

Пример 3. Грамматика, порождающая язык правильных скобочных выражений (язык Дика).

$$
\begin{aligned}
G_3:\ & S \to (S) && (1)\\
& S \to SS && (2)\\
& S \to \varepsilon && (3)
\end{aligned}
$$

Нетрудно понять логику построения правил этой грамматики. Смысл первого правила таков: заключив в скобки правильное скобочное выражение S, мы снова получим правильное скобочное выражение. Второе правило означает, что два правильных скобочных выражения, записанные одно за другим, дают новое правильное выражение. Наконец, по правилу (3) пустая цепочка считается правильным выражением. Если бы мы решили, что не следует разрешать пустые выражения, правило (3) можно было бы заменить на $S \to ()$.

Пример 4. Грамматика простых арифметических выражений. Единственным нетерминалом этой грамматики (он же начальный) будет *Выражение:*

$$G_4\text{:}\quad \textit{Выражение} \to \textit{Выражение} + \textit{Выражение} \mid \textit{Выражение} - \textit{Выражение} \mid \textit{Выражение} * \textit{Выражение} \mid \textit{Выражение} / \textit{Выражение} \mid a \mid b \mid c \mid (\ \textit{Выражение}\ ).$$

Такая грамматика порождает цепочки терминалов, являющиеся правильными арифметическими выражениями. Символы *a, b* и *c* в таких выражениях обозначают операнды, а «+», «–», «*», и «/» — знаки операций. Разрешаются круглые скобки (в том числе вложенные). Примеры правильных выражений: $(a+b)/(b*c)$, (a). Все операции двуместные, унарные плюс и минус не предусмотрены, поэтому, например, цепочка $-a$ не принадлежит языку, порождаемому этой грамматикой.

Выражение — одно из основных понятий языков программирования. В дальнейшем грамматикам выражений будет уделено немалое внимание. Чтобы запись этих грамматик была короче, заменим нетерминал *Выражение* на E (от *expression* — выражение), вернувшись тем самым к принятым раньше соглашениям об именовании нетерминалов. Тогда грамматика, которую обозначим G_5, запишется следующим образом:

$$G_5\text{:}\ E \to E + E \mid E - E \mid E * E \mid E / E \mid a \mid b \mid c \mid (\ E\ ).$$

Очевидно, что она порождает тот же язык, что и грамматика G_4. А именно:

$$L(G_5) = L_5.$$

Еще несколько определений

В предыдущих примерах мы уже пользовались такими терминами как «вывод», «язык, порождаемый грамматикой». Теперь дадим формальные определения для этих и ряда других понятий.

Пусть имеются грамматика $G = (T, N, P, S)$ и цепочка α_1, составленная из терминалов и нетерминалов грамматики G, причем α_1 представима в виде $\alpha_1 = \gamma_1\alpha\gamma_2$, где γ_1, γ_2 — цепочки терминалов и нетерминалов грамматики G, α — непустая цепочка терминалов и нетерминалов грамматики G: $\gamma_1, \gamma_2 \in (T \cup N)^*$; $\alpha \in (T \cup N)^+$. Пусть также среди множества правил P грамматики G есть правило $\alpha \to \beta$. Тогда подцепочка α цепочки α_1 может быть заменена цепочкой β, в результате чего будет получена цепочка $\alpha_2 = \gamma_1\beta\gamma_2$. В этом случае говорят, что цепочка α_2 *непосредственно выводится* (порождается) из цепочки α_1 в грамматике G, что записывается следующим образом:

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2 \text{ или просто } \alpha_1 \Rightarrow \alpha_2,$$

если используемая грамматика очевидна.

Если имеется последовательность цепочек $\alpha_1, \alpha_2, \ldots, \alpha_n$ $(n > 1)$, таких что

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2 \ldots \underset{G}{\Rightarrow} \alpha_n, \tag{*}$$

то говорят, что цепочка α_n *нетривиально выводится* из α_1 в грамматике G (выводится за один или более шагов), что обозначается так:

$$\alpha_1 \overset{+}{\underset{G}{\Rightarrow}} \alpha_n \text{ или просто } \alpha_1 \overset{+}{\Rightarrow} \alpha_n.$$

Последовательность цепочек $\alpha_1, \alpha_2, \ldots, \alpha_n$ в этом случае называется **выводом** цепочки α_n из α_1 в грамматике G.

В дальнейшем выводом будем называть также запись, подобную (*). Используется также запись

$$\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_n \text{ или } \alpha_1 \overset{*}{\Rightarrow} \alpha_n,$$

означающая, что цепочка α_n *выводится* из α_1 в грамматике G (выводится *за ноль или более* шагов), то есть либо α_n совпадает с α_1, либо $\alpha_1 \overset{+}{\Rightarrow} \alpha_n$.

Сентенциальной формой грамматики G называется цепочка, выводимая из начального нетерминала грамматики G.

Цепочка α — есть сентенциальная форма грамматики G, если $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$.

Сентенцией (от *sentence* — предложение) грамматики G называется сентенциальная форма, состоящая только из терминальных символов.

Язык, порождаемый грамматикой, есть множество всех её сентенций.

Можно сказать, что сентенции грамматики — это предложения порождаемого ею языка.

Дерево вывода

В последующем рассмотрении предполагается, что мы имеем дело с грамматиками, все правила которых в своей левой части содержат единственный нетерминал. Именно такие грамматики, называемые контекстно-свободными, представляют для нас наибольший практический интерес.

Рассмотрим грамматику.

G_6: $S \rightarrow AB$ (1)

$A \rightarrow aA \mid a$ (2)

$B \rightarrow bB \mid b$ (3)

Выполним вывод цепочек в этой грамматике. Вначале используем правило (1):

$$S \Rightarrow AB.$$

Будем сопровождать процесс вывода построением дерева (рис. 2.2). Корнем дерева будет вершина, соответствующая начальному нетерминалу *S*. Дочерними вершинами корня будут вершины *A* и *B*, соответствующие правой части первого примененного правила (рис. 2.2*а*). Вершины *A* и *B* следуют в дереве слева направо в том же порядке, что и в правиле (1): слева — *A*, справа — *B*.

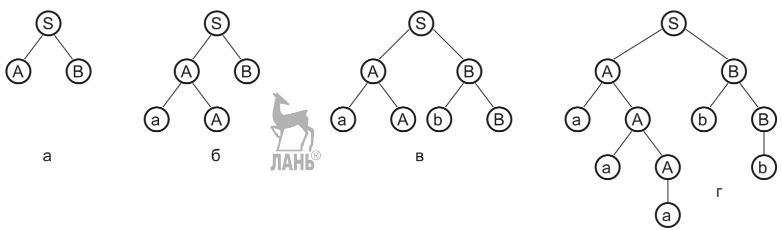


Рис. 2.2. Построение дерева вывода

Продолжая вывод, мы можем выбрать, как правило для нетерминала *A* — правило (2), так и правило для *B* — правило (3). Используем вначале первую часть правила (2) ($A \rightarrow aA$):

$$S \Rightarrow AB \Rightarrow aAB$$

и продолжим построение дерева (рис. 2.2*б*). Теперь выполним подстановку вместо нетерминала B цепочки bB по правилу (3):

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aAbB,$$

добавив к имеющейся вершине B дочерние вершины b и B (рис. 2.2*в*). Еще раз применим правило $A \to aA$ и, чтоб получить цепочку, состоящую только из терминалов, выполним подстановки по правилам $A \to a$ и $B \to b$. После этого вывод приобретет следующий вид:

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aAbB \Rightarrow aaAbB \Rightarrow aaabB \Rightarrow aaabb,$$

а получившееся дерево показано на рисунке 2.2*г*.

Получившееся дерево называется *деревом вывода, деревом разбора* или *синтаксическим деревом*[49]. Его корень — начальный символ грамматики, внутренние вершины — нетерминалы, листья дерева (концевые вершины) — терминалы. Обход листьев дерева слева направо дает цепочку терминалов, выведенную из начального символа грамматики (сентенцию)[50].

Нетрудно заметить, что построенное нами дерево будет соответствовать и другим выводам цепочки $aaabb$ в грамматике G_6. Вот один из них:

$$S \Rightarrow AB \Rightarrow AbB \Rightarrow aAbB \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb.$$

Рассмотрение дерева вместо вывода позволяет игнорировать порядок применения правил, если он не важен.

Задача разбора

Задача разбора состоит в восстановлении дерева вывода для заданной сентенции.

Разбор — это построение вывода для заранее заданной цепочки. Другими словами, разбор — это тот же вывод, прослеженный в обратном порядке. Последовательность сентенциальных форм, приводящая к цепочке терминалов (сентенции, предложению языка, по-

---

[49] К сожалению, в литературе на русском языке существует путаница в терминах. Одни авторы (и их переводчики) [Грис, 1975] считают «дерево разбора» и «синтаксическое дерево» синонимами, другие [Ахо, 2001] придают понятию «синтаксическое дерево» иной смысл. Я буду придерживаться первого варианта, поскольку термины «разбор» и «синтаксический анализ» означают одно и то же, и было бы странно их различать, говоря о деревьях.

[50] Могут рассматриваться и деревья вывода, листьями которых являются как терминалы, так и нетерминалы. В этом случае обход листьев дает сентенциальную форму грамматики. Однако такие деревья нам не потребуются.

рождаемого грамматикой), определяет структуру этой цепочки. Дерево вывода представляет структуру цепочки нагляднее и независимо от последовательности применения правил.

Результатом решения задачи разбора в случае, если удалось восстановить дерево для заданной терминальной цепочки, является выявление структуры этой цепочки. Построенное дерево называется *деревом разбора*. Успешное восстановление дерева разбора для заданной цепочки означает, что эта цепочка есть правильное предложение языка, порождаемого грамматикой. Наоборот, если для некоторой цепочки терминалов дерево разбора в данной грамматике построить невозможно, это значит, что цепочка не принадлежит порождаемому грамматикой языку.

Для чего надо решать задачу разбора

Разбор (по-английски — parsing) называют также распознаванием или синтаксическим анализом. Синтаксический анализ имеет две цели — выяснение принадлежности цепочки языку и выявление ее структуры.

Работа любого транслятора основана на распознавании структуры предложений транслируемого языка. Синтаксический анализ — обязательная фаза в работе компиляторов и интерпретаторов языков программирования. Синтаксический анализатор — это часть транслятора, составляющая его основу.

Только распознавая структуру входной программы, определяя наличие или отсутствие отдельных ее частей — описаний, операторов, выражений и подвыражений — транслятор может выполнить работу по переводу программы на другой язык. Часто трансляторы в явном виде строят дерево программы, которое представляется внутренними динамическими структурами данных транслятора, а затем используется при формировании эквивалентной выходной программы. Если в ходе распознавания дерево вывода не строится, оно присутствует неявно, отражаясь в последовательности выполняемых синтаксическим анализатором действий.

Рассмотрению способов решения задачи разбора — синтаксического анализа будет посвящена большая часть этой главы.

Домино Де Ремера

Де Ремер (De Remer F. L.) предложил наглядную интерпретацию задачи разбора, представив ее как игру в своеобразное домино.

Играющий располагает «костями» домино нескольких типов. Типов столько, сколько правил в грамматике. Каждое правило дает один тип пластинки. Типы домино для грамматики G_6 показаны на рисунке 2.3. Считается, что «костяшек» каждого типа имеется сколько необходимо.

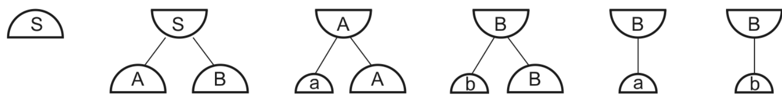


Рис. 2.3. Домино Де Ремера для грамматики G_6

Верхняя часть каждого домино соответствует левой части правила грамматики, нижняя — правой. Верхняя и нижние пластинки соединены «резиновыми» нитями. Пластинки можно приставлять друг к другу плоскими сторонами полукруга, если на них записаны одинаковые символы. Фигуры домино нельзя переворачивать, и нельзя менять порядок следования символов (перекрещивать нити).

В начале игры в верхней части поля помещается полукруг, обращенный выпуклостью вверх, в котором записан начальный нетерминал грамматики. В нижней части игрового поля в полукругах, обращенных плоской частью вверх, размещаются терминальные символы распознаваемой цепочки. На рисунке 2.4 показана начальная конфигурация игры для цепочки *aaabb*.

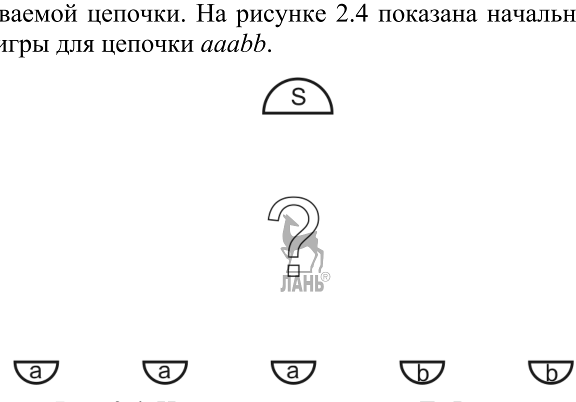


Рис. 2.4. Начало игры в домино Де Ремера

Цель состоит в том, чтобы соединить с помощью имеющихся фигур символы терминальной цепочки и начальный нетерминал. Полученная конфигурация домино для цепочки *aaabb* и грамматики G_6 (набор домино на рис. 2.3) показана на рисунке 2.5.

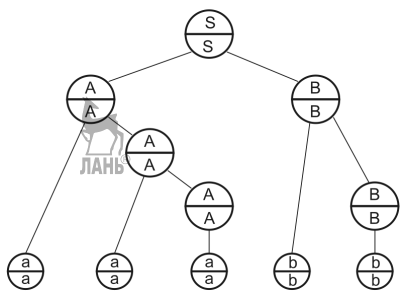


Рис. 2.5. Дерево разбора, построенное из домино Де Ремера

Разновидности алгоритмов разбора

Имея в виду интерпретацию Де Ремера, можно представить себе и различные подходы к решению задачи разбора. Если подбор костей удается осуществлять так, что однажды поставленную кость никогда не придется убирать, мы имеем дело с детерминированным алгоритмом разбора — выбор применяемого правила грамматики всегда однозначен. Если принятые решения о выборе типа домино приходится отменять — алгоритм работает с возвратами, он недетерминирован. Детерминированные алгоритмы эффективней и, конечно, всегда существует стремление найти и использовать такой алгоритм для синтаксического анализа.

Если дерево строится сверху вниз от начального нетерминала в сторону терминальной цепочки, алгоритм относится к классу нисходящих; от цепочки в сторону корня дерева — восходящий (рис. 2.6). Можно вначале подбирать домино для левых символов терминальной цепочки, а можно вначале для правых — соответственно говорят о левосторонних или правосторонних алгоритмах разбора.

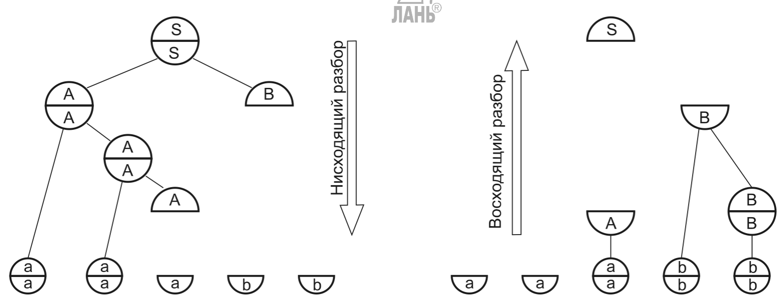


Рис. 2.6. Начало нисходящего и восходящего разбора в грамматике G_6

Эквивалентность и однозначность грамматик

Возьмем для примера грамматику арифметических выражений

$$G_5\colon\ E \to E + E \mid E - E \mid E * E \mid E / E \mid a \mid b \mid c \mid (E).$$

Рассмотрим деревья вывода терминальных цепочек в этой грамматике. На рисунке 2.7 показаны два различных дерева разбора выражения *a*+*b**c*. Возможность построить эти деревья убеждает в том, что цепочка *a*+*b**c* действительно принадлежит языку арифметических выражений.

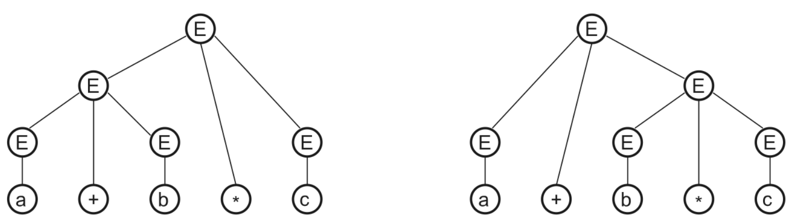


Рис. 2.7. Деревья разбора цепочки $a+b*c$ в грамматике G_5

Но два разных дерева дают две разные трактовки структуры этой цепочки. Дерево слева объединяет $a+b$ в одно подвыражение, которое затем участвует в роли операнда в операции умножения. Такая трактовка не соответствует общепринятому приоритету арифметических операций — умножение должно выполняться раньше сложения. Дерево справа представляет структуру, соответствующую правильному порядку выполнения операций.

Грамматика G называется однозначной, если любой сентенции $x \in L(G)$ *соответствует единственное дерево вывода.*

Грамматика G_5 неоднозначна, и это, безусловно, ее серьезный недостаток, который не позволяет применить ее на практике для определения языка выражений, поскольку эта грамматика не позволяет однозначным образом выявить структуру выражения.

Попробуем построить однозначную грамматику выражений. Для этого используем еще один нетерминал, обозначив его X.

$$G_7\colon\ E \to X \mid E + X \mid E - X \mid E * X \mid E / X$$
$$X \to a \mid b \mid c \mid (E)$$

Содержательно X можно понимать как операнд выражения. Теперь для цепочки $a+b*c$ можно построить единственное дерево разбора. Оно показано на рисунке 2.8 слева. Можно убедиться, что в грамма-

тике G_7 единственное дерево вывода соответствует любому правильному выражению. Грамматика G_7 однозначна.

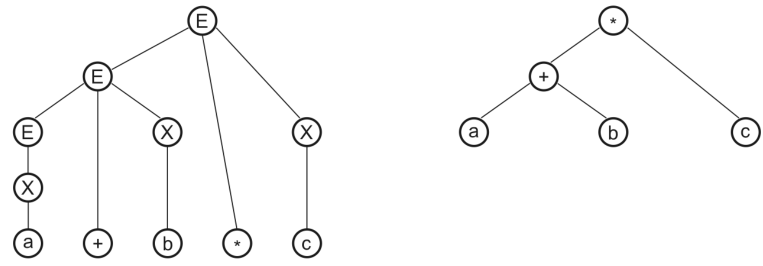


Рис. 2.8. Дерево разбора выражения $a+b*c$ в грамматике G_7

Справа на рисунке показано редуцированное (упрощенное) дерево того же выражения в грамматике G_7. Такие деревья могут служить для представления структуры выражений в трансляторе. Будем называть их *семантическими деревьями*[51]. Семантическое дерево может быть получено из дерева разбора устранением нетерминальных вершин и помещением знаков операций во внутренние вершины, в то время как операнды остаются листьями дерева.

Несмотря на однозначность, G_7 непригодна для использования в трансляторе, поскольку приписывает выражениям неподходящую структуру. Уже на примере цепочки $a+b*c$ (см. рис. 2.8) видно, что операции и операнды связываются неправильно. Нетрудно заметить, что G_7 предполагает выполнение операций без учета их приоритета в порядке слева направо.

Исправить положение можно, определив нетерминалы для двух категорий подвыражений — слагаемых и множителей. После этого выражение представляется как последовательность слагаемых, разделенных знаками плюс и минус. В свою очередь слагаемые образуются из элементарных операндов — множителей, соединенных знаками

[51] Как уже говорилось, существует несогласованность в использовании терминов, относящихся к синтаксическим и семантическим деревьям. В этой книге мы будем, следуя [Рейуорд-Смит, 1988], придерживаться названия «семантическое дерево». В то же время в издании [Ахо, 2001] в этом случае говорится о синтаксическом дереве, что противоречит терминологии классической монографии [Грис, 1975] и, по моему мнению, создает путаницу, поскольку трудно видеть разницу между синтаксическим деревом и деревом разбора.

умножения и деления. Слагаемые обозначим T (от term — элемент), множители — M (от multiplier).

G_8: $E \rightarrow T \mid E + T \mid E - T$

$T \rightarrow M \mid T * M \mid T / M$

$M \rightarrow a \mid b \mid c \mid (E)$

Дерево вывода цепочки $a+b*c$ в грамматике G_8 показано на рисунке 2.9. Эта грамматика однозначна и приписывает арифметическим выражениям структуру, соответствующую правильному порядку выполнения операций.

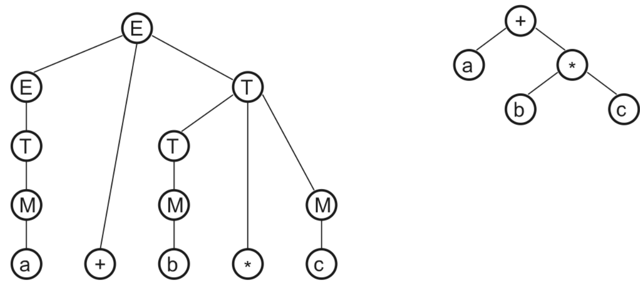


Рис. 2.9. Дерево выражения $a+b*c$ в грамматике G_8

Все три рассмотренные выше грамматики выражений (G_5, G_7, G_8), хотя и различаются и обладают разными свойствами, задают один и тот же язык.

Грамматики называются **эквивалентными***, если порождают один и тот же язык.*

Грамматики G_5, G_7, G_8 эквивалентны, поскольку $L(G_5) = L(G_7) = L(G_8)$.

Иерархия грамматик Н. Хомского

Н. Хомский предложил разделение порождающих грамматик на 4 типа в зависимости от вида их правил.

Тип 0. *Произвольные грамматики.* На вид их правил не накладывается каких-либо ограничений. Правила имеют вид:

$$\alpha \rightarrow \beta,$$

где α и β — цепочки терминалов и нетерминалов. Цепочка α не должна быть пустой.

Тип 1. *Контекстно-зависимые грамматики.* Правила таких грамматик имеют вид:

$$\alpha A\beta \to \alpha\gamma\beta,$$

где α, β, γ — цепочки терминалов и нетерминалов; A — нетерминальный символ. Такой вид правил означает, что нетерминал A может быть заменен цепочкой γ только в контексте, образуемом цепочками α и β.

Тип 2. Контекстно-свободные грамматики. Их правила имеют вид:

$$A \to \gamma,$$

где A — нетерминал; γ — цепочка терминалов и нетерминалов. Характерная особенность — в левой части правил всегда один нетерминальный символ.

Тип 3. *Автоматные грамматики.* Все правила автоматных грамматик имеют одну из трех форм:

$$A \to aB,$$
$$A \to a,$$
$$A \to \varepsilon,$$

где A, B — нетерминалы; a — терминал; ε — пустая цепочка. Автоматные грамматики называют также *регулярными*.

Как можно видеть, грамматики типа 1 являются частным случаем грамматик типа 0, грамматики типа 2 — частный случай контекстно-зависимых грамматик, автоматные — частный случай контекстно-свободных. То есть грамматика типа 3 является и грамматикой типа 2, и типа 1, и типа 0. Однако в дальнейшем, если не оговорено особо, будет иметься в виду что, например, контекстно-свободной называется грамматика, не являющаяся автоматной.

Языки, порождаемые грамматиками типа 0–3, называются соответственно языками без ограничений, контекстно-зависимыми, контекстно-свободными и автоматными (регулярными) языками. Но контекстно-свободным языком называют язык, для которого существует порождающая его контекстно-свободная, но не автоматная грамматика. Такой же подход применяется к контекстно-зависимым языкам и языкам без ограничений.

Примеры грамматик различных типов

Рассмотрим грамматику G_2, порождающую язык $L_2 = \{a^n b^n c^n \mid n \geq 0\}$.

G_2: $S \to aSBc$ (тип 2)
$S \to abc$ (тип 2)
$cB \to Bc$ (тип 0)
$bB \to bb$ (тип 1)
$S \to \varepsilon$ (тип 3)

Справа около каждого правила помечен тип грамматики, к которой оно может быть отнесено. Типом грамматики естественно считать минимальный из типов ее правил. Следовательно, грамматика G_2 — это грамматика типа 0 — произвольная. Утверждается, однако, что может быть построена контекстно-зависимая грамматика (типа 1), порождающая тот же язык, что и G_2. Проверку этого утверждения предоставлю читателям.

Примером контекстно-свободной грамматики может служить грамматика арифметических выражений. С помощью контекстно-свободных грамматик задается и синтаксис языков программирования. Грамматики этого класса будут подробно обсуждаться и в дальнейшем, сейчас же возьмем конкретный пример.

G_9: $N \to a$ (1)
$N \to Na$ (2)
$N \to Nb$ (3)

Это грамматика типа 2, поскольку правила (2) и (3) относятся именно к этому типу. Рассмотрим язык $L_9 = L(G_9)$, порождаемый этой грамматикой. Цепочка a принадлежит L_9 по правилу (1). Если к правильному предложению N языка приписать справа символ a, то снова получится правильное предложение (по правилу (2)). Аналогично, приписывание к N символа b снова дает предложение языка L_9. Принадлежащие языку L_9 цепочки начинаются символом a, за которым могут следовать a и b в произвольном порядке. Если под a понимать любую латинскую букву, а b воспринимать как цифру, то G_9 можно считать грамматикой идентификаторов. Она порождает последовательности букв и цифр, начинающиеся с буквы, которые используются в языках программирования в роли имен переменных, типов и т. д.

Опираясь на такую содержательную трактовку G_9 и L_9, попытаемся сконструировать автоматную грамматику, порождающую язык идентификаторов.

Первое правило грамматики G_9 может быть сохранено. Оно, во-первых, соответствует одному из допустимых видов правил автоматных грамматик, во-вторых, определяет, что идентификатор, состоящий из одного символа, может быть только буквой.

$N \to a.$ (1)

Нетерминал N — это начальный символ нашей грамматики. Он и обозначает само понятие «идентификатор». Как синоним термина «идентификатор» будем использовать также слово «имя». Можно

считать, что название N происходит от Name — имя. Обозначим B часть идентификатора, которая может следовать за первой буквой. Тогда можно записать правило (2):

$N \to aB.$ (2)

Запишем правила для B. «Хвост» может быть буквой или цифрой:

$B \to a,$ (3)

$B \to b.$ (4)

Если B — это «хвост» идентификатора, то, записав его за буквой или цифрой, снова получим правильный «хвост»:

$B \to aB,$ (5)

$B \to bB.$ (6)

Обозначим сконструированную грамматику G_{10}. Она автоматная, поскольку все ее правила удовлетворяют ограничениям автоматных грамматик.

G_{10}: $N \to a$ (1)

$N \to aB$ (2)

$B \to a$ (3)

$B \to b$ (4)

$B \to aB$ (5)

$B \to bB$ (6)

Грамматика G_{10} эквивалентна G_9, поскольку порождает тот же язык: $L(G_{10}) = L(G_9)$. Этот язык — язык идентификаторов — следует считать автоматным. Нетрудно увидеть возможности упрощения грамматики G_{10}. Отложим, однако, на некоторое время такое упрощение.

Автоматные грамматики и языки

Рассмотрим автоматные грамматики и языки подробнее, имея целью построение алгоритмов распознавания этого класса языков.

Граф автоматной грамматики

Для каждой автоматной грамматики можно построить направленный граф по следующим правилам:

1. Каждому нетерминальному символу грамматики ставится в соответствие вершина графа, которая помечается этим символом.
2. При наличии правил вида

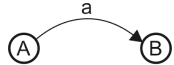
$$A \to a$$

добавляется дополнительная вершина, которая помечается символом K.

3. Каждое правило вида

$$A \to aB$$

порождает дугу графа, ведущую из вершины A в вершину B.

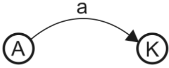


Дуга помечается символом a.

4. Каждое правило вида

$$A \to a$$

порождает дугу графа, ведущую из вершины A в вершину K.



Дуга помечается символом a.

5. Вершина, соответствующая начальному нетерминалу, помечается стрелкой.

6. Вершина K и вершины, соответствующие нетерминалам, для которых есть правило

$$A \to \varepsilon,$$

помечаются как конечные. Мы будем изображать их двойным кружком.

Построим граф автоматной грамматики G_{10} (рис. 2.10). Двум нетерминалам этой грамматики будут соответствовать вершины N и B (п. 1). Поскольку в грамматике есть несколько правил, в правой части которых записан единственный терминал, добавим вершину K (п. 2).

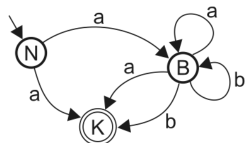


Рис. 2.10. Граф автоматной грамматики G_{10}

Соединим вершины дугами, как это предписывается п. 3 и п. 4. Вершину N пометим стрелкой как начальную (п. 5).

Граф автоматной грамматики может использоваться для порождения цепочек языка. Любой путь из начальной вершины графа в одну из конечных вершин порождает цепочку терминалов, соответствующую проходимым дугам. Эта цепочка принадлежит языку, порождаемому грамматикой. И, наоборот, для любой сентенции грамматики можно найти путь, ведущий из начальной вершины в одну из конечных и проходящий по дугам, помеченным символами этой сентенции.

Грамматика G_{10} порождает язык идентификаторов. Нетрудно убедиться, что для любого идентификатора найдется путь из вершины N в вершину K, а любой путь из N в K соответствует правильному идентификатору.

Граф автоматной грамматики идентичен диаграмме переходов конечного автомата — абстрактного устройства, являющегося моделью определенного класса реальных автоматических устройств и объектом изучения теории автоматов.

Конечные автоматы

Конечным автоматом (КА) называется пятерка:

$$A = (N, T, P, S, F),$$

где N — конечное множество состояний автомата.

T — входной алфавит — конечное множество символов.

P — функция переходов автомата (в общем случае неоднозначная), отображающая множество пар состояние–входной символ в множество состояний[52].

S — начальное состояние. $S \in N$.

F — множество конечных (финитных) состояний. $F \subseteq N$.

Конечный автомат действует следующим образом. Вначале он находится в состоянии S. На вход КА поступают символы, принадлежащие входному алфавиту. Последовательность входных символов образует входную цепочку. Находясь в некотором состоянии и получив на вход очередной символ, автомат переходит в следующее состояние, определяемое значением функции переходов для данной пары символ–состояние, и считывает очередной символ. В общем случае функция переходов может определять переход в несколько состояний для данной пары символ–состояние. В этом случае говорят

[52] Вместо того чтоб считать функцию переходов неоднозначной, можно было бы говорить, что ее значениями для данной пары символ–состояние являются не отдельные состояния, а множества состояний.

о недетерминированным конечном автомате (НКА). Автомат останавливается, когда заканчиваются символы на его входе.

Если, прочитав входную цепочку α, автомат остановился в некотором состоянии B, говорят, что цепочка α перевела автомат из начального состояния в состояние B. Если B — одно из конечных состояний ($B \in F$), то говорят, что автомат принимает (допускает) цепочку α.

Множество всех цепочек, переводящих конечный автомат A из начального в одно из конечных состояний (*множество цепочек, принимаемых КА*)*, образует язык L*(*A*)*, принимаемый* (*допускаемый*) *КА.*

Язык, порождаемый автоматной грамматикой G, совпадает с языком, принимаемым соответствующим конечным автоматом A.

$$L(G) = L(A)$$

Как мы уже видели, КА может задаваться с помощью диаграммы переходов. Например, граф автоматной грамматики G_{10}, показанный на рисунке 2.10, может считаться диаграммой переходов автомата A_{10}. При этом $L(G_{10}) = L(A_{10})$.

Преобразование недетерминированного конечного автомата (НКА) в детерминированный конечный автомат (ДКА)

То обстоятельство, что при переходе от автоматной грамматики к КА мы получаем в общем случае НКА, затрудняет его использование в роли распознавателя автоматного языка. Недетерминированность автомата выражается в том, что для некоторых вершин его диаграммы переходов имеется несколько дуг, выходящих из этих вершин и помеченных одним и тем же символом. Так, автомат, изображенный на рисунке 2.10, является недетерминированным. Из вершины N исходят две дуги, помеченные символом a, из вершины B — по две дуги, помеченных символами a и b.

Было бы крайне желательно иметь возможность строить для автоматной грамматики детерминированный конечный автомат.

Теорема Клини. *Для каждого НКА можно построить ДКА, допускающий тот же язык.*

Рассмотрим алгоритм построения ДКА, эквивалентного данному НКА. Для иллюстрации алгоритма будем применять его к НКА A_{10} (см. рис. 2.10), принимающему язык идентификаторов.

1. Пусть исходный НКА имеет k состояний. Для построения ДКА возьмем $2^k - 1$ состояний, каждое из которых соответствует одному

элементу множества всех подмножеств состояний исходного автомата, кроме пустого множества.

Автомат A_{10} имеет три ($k = 3$) состояния: N, B и K. У нового автомата будет $2^3 - 1 = 7$ состояний. Они соответствуют таким множествам состояний исходного НКА: $\{N\}$, $\{B\}$, $\{K\}$, $\{N, B\}$, $\{B, K\}$, $\{N, K\}$, $\{N, B, K\}$. Будем обозначать состояния нового автомата просто последовательностями букв: N, B, K, NB, BK, NK, NBK.

2. Из каждого состояния S нового автомата направим *не более чем один переход*, *помеченный данным символом,* в такое состояние, которое соответствует множеству состояний НКА, в которые есть переходы по этому символу хотя бы из одного состояния НКА, образующего S.

   У НКА A_{10} из состояния N есть переход по символу a в состояния B и K (см. рис. 2.10). Следовательно, из состояния N нового автомата дугу, помеченную символом a, направляем в состояние BK.

   Рассматривая состояние NB нового автомата, выясняем, что переходы из состояния N по символу a в исходном автомате есть в состояния B и K, из состояния B исходного автомата — также в состояния B и K. Направляем дугу, помеченную a, из состояния NB в состояние BK (рис. 2.11). Перехода по символу b из состояния N в исходном автомате нет. Из состояния B исходного автомата есть переходы, помеченные b, в состояния B и K. Направляем дугу b из состояния NB в состояние BK.

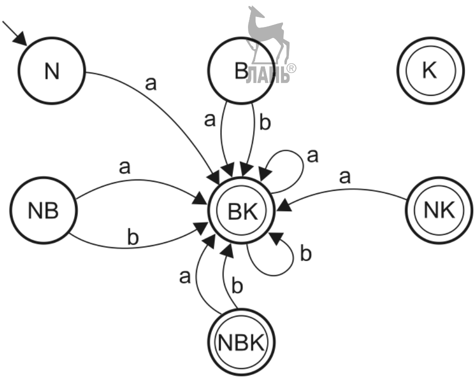


Рис. 2.11. Детерминированный конечный автомат A_{11}, эквивалентный недетерминированному автомату A_{10}

Аналогично формируем переходы из других состояний нового автомата A_{11}, который будет эквивалентен A_{10}. В нашем примере оказывается, что все дуги, помеченные как символом a, так и символом b, ведут в состояние BK.

3. В качестве начального состояния ДКА отметим состояние, имеющее то же обозначение, что и начальное состояние исходного НКА. Как конечные отметим все состояния ДКА, в которые входит хотя бы одно из конечных состояний исходного НКА.

В нашем примере начальным будет состояние N. Конечными состояниями ДКА будут все состояния, включающие состояние K исходного автомата, то есть K, BK, NK, NBK (см. рис. 2.11).

Получившийся автомат является детерминированным (из любого состояния исходит не более одной дуги, помеченной данным символом) и принимает тот же язык, что и исходный недетерминированный автомат.

Детерминированный автомат A_{11} имеет больше состояний, чем исходный НКА A_{10}. Нетрудно, однако, увидеть возможности упрощения получившегося ДКА. Большинство его состояний (B, K, NB, NK, NBK) недостижимы из начального состояния, поэтому могут быть отброшены. Получающийся после этого ДКА A_{12} показан на рисунке 2.12. Обозначение состояния BK упрощено, оно снова названо просто B. Этот автомат не только детерминирован, но и проще исходного НКА A_{10}.

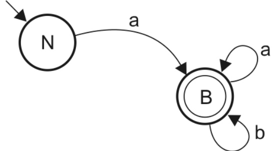


Рис. 2.12. Минимальный ДКА A_{12}, распознающий идентификаторы

По диаграмме переходов получившегося автомата можно снова записать автоматную грамматику, порождающую язык идентификаторов, эквивалентную грамматике G_{10}, но содержащую меньше правил:

G_{12}: $N \to aB$,

$B \to aB \mid bB \mid \varepsilon$.

Кстати, смысл нетерминала B в новой грамматике сохранился — это «хвост», завершающий идентификатор.

Детерминированный конечный автомат можно рассматривать как распознаватель автоматного языка — устройство, с помощью которо-

го просто и эффективно решается задача разбора для автоматной грамматики. В связи с этим, наряду с автоматами принимающими (допускающими) некоторый язык, будем говорить об автоматах, *распознающих* язык.

Для любого автоматного языка можно построить детерминированный конечный автомат, распознающий этот язык.

Задача получения возможно более простого ДКА также имеет общее решение.

Для любого автоматного языка можно построить единственный ДКА, распознающий этот язык и имеющий минимально возможное число состояний.

С алгоритмом построения ДКА с минимальным числом состояний можно познакомиться в книгах [Ахо, 2008], [Карпов, 2002], [Хопкрофт, 2002].

Таблица переходов детерминированного конечного автомата

Наряду с представлением графом, функция переходов ДКА может быть задана таблицей, что, безусловно, больше подходит для программной реализации конечного автомата (табл. 2.1). Рассмотрим таблицу переходов ДКА A_{12}, распознающего язык идентификаторов.

Таблица 2.1. Таблица переходов конечного автомата A_{12}

Состояние	**Символ**	
	a	***b***
N	*B*	*E*
B	*B*	*B*
E	*E*	*E*

В таблице записано состояние, в которое переходит автомат, находясь в состоянии, соответствующем данной строке таблицы и, получив входной символ, обозначенный в соответствующем столбце. Конечное состояние автомата ***B*** помечено в таблице жирным шрифтом.

Наряду с состояниями *N* и *B* предусмотрено дополнительное состояние *E* — состояние ошибки. Это сделано для того, чтобы функция переходов была определена для всех возможных пар символ–состояние. Иначе переход из состояния *N* при поступлении на вход символа *b* был бы не определен. Попав в состояние *E*, автомат остается в нем. Состояние *E* не является конечным. На практике при программной реализа-

ции, кроме символов входного алфавита, потребуется, скорее всего, определить реакции автомата и на любые другие символы, которые, очевидно, должны переводить автомат в состояние *E*.

Программная реализация автоматного распознавателя

В листинге 2.1 приведен эскиз программы, моделирующей работу детерминированного конечного автомата. Эта программа и является универсальным распознавателем (синтаксическим анализатором) автоматных языков. В ней есть лишь некоторые условности: предполагается, что состояния обозначаются латинскими буквами (*S* — начальное состояние), а входной алфавит — малые латинские буквы. Не конкретизированы также способы считывания символов и проверки их наличия на входе, а также то, как автомат реагирует на принятие или непринятие входной цепочки — эти части программы записаны по-русски.

Листинг 2.1. Универсальный распознаватель автоматных языков

```
type
   tCondition = (S, A, B, C, …, E, …); { Состояния }
   tAlpha     = 'a'..'z';              { Алфавит   }
  tJump      = array [tCondition, tAlpha] of tCondition;
     { Таблица переходов }
   tFinish    = set of tCondition;
      { Тип множества конечных состояний }
var
   Cond : tCondition;        { Текущее состояние   }
   Ch   : tAlpha;            { Входной символ      }
   P    : tJump;             { Функция переходов   }
   Fin  : tFinish;           { Конечные состояния }
…
{ Здесь задаются значения P и Fin }
…
Cond := S;
while Есть символы do begin
   Читать(Ch);
   Cond := P[Cond, Ch]
end;
if Cond in Fin then
   Цепочка принята
else
   Цепочка не принята
```

Дерево разбора в автоматной грамматике

Говоря о задаче синтаксического анализа, мы сводили ее к построению дерева разбора. Между тем, в предыдущих разделах на роль распознавателя автоматных языков предложен конечный автомат, который дерево не строит. Нет ли здесь противоречия? Нет. Дерево разбора цепочки в автоматной грамматике может быть однозначно построено, если известна последовательность переходов конечно-автоматного распознавателя. То есть распознающий автомат не только дает ответ на вопрос о принадлежности цепочки языку, но и позволяет выявить структуру цепочки. Структура при этом представлена последовательностью переходов автомата.

Рассмотрим, какой вид имеет дерево разбора терминальной цепочки в автоматной грамматике. Из трех видов правил автоматной грамматики правила вида $A \to a$ и $A \to \varepsilon$ могут быть использованы в процессе порождения цепочки ровно один раз, после чего процесс порождения заканчивается. Все остальные подстановки выполняются по правилам вида $A \to aB$. Каждая такая подстановка приводит к появлению в дереве новой внутренней вершины, помеченной нетерминалом. Её левая дочерняя вершина помечается терминалом (рис. 2.13).

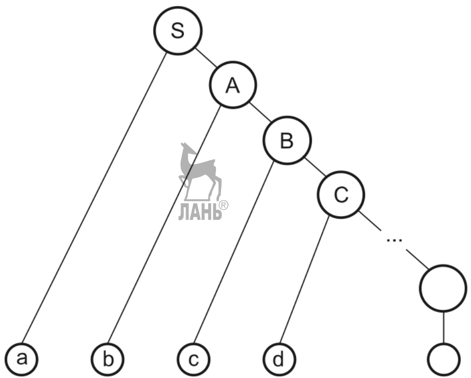


Рис. 2.13. Дерево разбора в автоматной грамматике

Если для грамматики типа 3 построен ДКА, то входная цепочка однозначно определяет последовательность проходимых автоматом состояний и дерево вывода.

Пример автоматного языка

Рассмотрим язык целых чисел со знаком. Примеры правильно записанных чисел:

```
177  +22   -1   0  02
```

Построим конечный автомат, который распознает этот язык. Зададим этот автомат с помощью диаграммы переходов. Это диаграмма будет служить также и формальным определением самого языка.

Начальное состояние автомата обозначим *S* (рис. 2.14). Находясь в этом состоянии, автомат ожидает символ, с которого может начинаться запись числа. Это знаки «+», «–» и цифры. Соответственно, из состояния *S* должны исходить дуги, помеченные этими символами. Десять дуг, помеченных цифрами от 0 до 9, заменим одной, пометив ее символом *ц*. После того как принят знак числа, автомат должен перейти в состояние, в котором он ожидает первую цифру. Обозначим такое состояние *A*. Таким образом, переходы по символам «+» и «–» ведут из состояния *S* в состояние *A*.

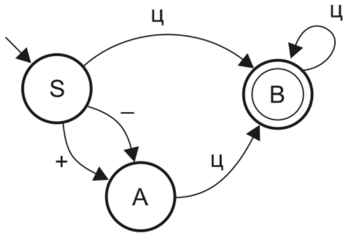


Рис. 2.14. Конечный автомат, распознающий целые числа со знаком

Если, находясь в начальном состоянии *S*, автомат получил цифру, он должен перейти в состояние (обозначим его *B*), в котором могут быть приняты последующие цифры, если они есть. Из состояния *A* при получении цифры автомат также переходит в состояние *B*. Состояние *B* следует пометить как конечное, поскольку переход в это состояние означает, что на вход автомата поступила правильная запись целого числа. Дуга, помеченная символом *ц*, ведущая из состояния *B* в него же, позволяет автомату принять вторую и последующие цифры числа, если они есть.

По диаграмме переходов можно записать и грамматику, порождающую язык целых чисел со знаком. Каждой дуге соответствует прави-

ло. Конечные состояния порождают правила с пустой цепочкой в правой части.

$S \rightarrow +A \mid -A \mid цB$

$A \rightarrow цB$

$B \rightarrow цB \mid \varepsilon$

На практике, как уже говорилось, приходится учитывать возможность поступления на вход автомата не только символов входного алфавита, но и любых других символов. В этой ситуации можно предполагать, что из любого состояния исходит дуга, ведущая в состояние ошибки *E* (рис. 2.15). Кроме того, удобно считать, что входная цепочка всегда завершается специальным символом «конец текста», который обозначают ⊥.

При использовании символа ⊥ к автомату следует добавить состояние *K*, которое будет единственным конечным, и в которое из «бывших» конечных состояний будут направлены дуги, помеченные ⊥.

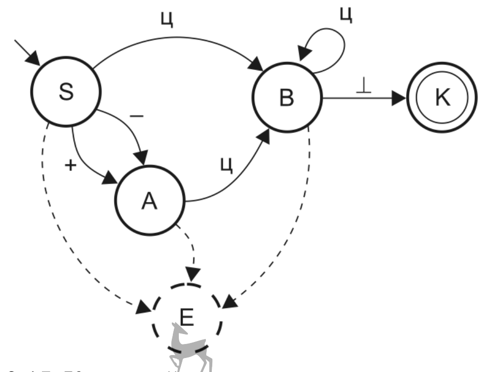


Рис. 2.15. Конечный автомат с состоянием ошибки и дополнительным конечным состоянием

Не составило бы труда заполнить для полученного автомата таблицу переходов и использовать универсальный распознаватель (см. листинг 2.1), моделирующий поведение ДКА. Но здесь мы рассмотрим другую возможность.

Пользуясь диаграммой переходов (см. рис. 2.14 и 2.15), напишем программу, которая ведет себя подобно конечному автомату, но не моделирует его поведение напрямую.

В начале работы, то есть, находясь в исходном состоянии, программа считывает первый символ входной цепочки и проверяет его допустимость:

```
Читать(Символ);
if not (Символ in ['+', '-', '0'.. '9']) then
   Ошибка
```

Будем считать, что обращение к процедуре `Ошибка` останавливает работу распознавателя. Далее (по-прежнему оставаясь в начальном состоянии) проверяем, какой именно из допустимых символов поступил на вход. Если это знак, читаем следующий символ, переходя к состоянию *A* и предусматривая действия, которые автомат выполняет, находясь в этом состоянии:

```
else if Символ in ['+', '-'] then begin
   Читать(Символ);
   { Состояние A }
   if Символ in ['0'..'9'] then
      Читать(Символ)
   else
      Ошибка
   end
else { Символ - цифра (в состоянии S) }
   Читать(Символ);
{ Состояние B }
```

Если в состоянии *S* поступил символ цифры, считывается следующий входной символ и происходит переход к состоянию *B*. После выполнения приведенного выше фрагмента программа или останавливается по причине ошибки, или приходит в состояние, аналогичное состоянию *B* конечного автомата. Находясь в состоянии *B*, автомат должен принимать все поступившие на вход цифры, оставаясь при этом в состоянии *B*:

```
{ Состояние B }
while Символ in ['0'.. '9'] do
   Читать(Символ);
```

Выход из цикла происходит, если очередной считанный символ — не цифра. Если это символ ⊥ — «конец текста», то входная цепочка закончена и автомат переходит в состояние *K* (см. рис. 2.15), принимая входную цепочку. Если цикл прекращен по причине поступления символа, отличного от цифры и ⊥, автомат переходит в состояние ошибки:

```
if Символ = ⊥ then
   Цепочка принята
else
   Ошибка;
```

Как видим, распознаватель автоматного языка, каким и является наша программа, можно написать, не прибегая к прямому моделированию поведения конечного автомата с использованием таблицы переходов. Такой подход может иметь свои преимущества. Технология программирования распознавателя оказывается довольно простой: программа пишется по диаграмме переходов ДКА, которая исполняет роль схемы алгоритма.

Синтаксические диаграммы автоматного языка

В построенной программе-распознавателе состояния конечного автомата не фигурировали явно. Они просто соответствовали некоторым точкам программы. В противоположность состояниям, символы, которыми помечены дуги диаграммы переходов, явно используются в распознавателе.

Устраним с диаграммы переходов обозначения состояний (заглавные буквы и кружки). Те места диаграммы, где были состояния автомата, превращаются в точки ветвления и соединения дуг. Терминальные символы, отмечающие переходы-дуги, наоборот разместим в кружках на этих дугах. Результат такого преобразования для диаграммы переходов автомата, распознающего целые числа, или, что то же самое, для графа автоматной грамматики, порождающей целые числа, показан на рисунке 2.16*а*. Полученный граф носит название *синтаксической диаграммы* автоматного языка или автоматной грамматики.

Синтаксическая диаграмма может использоваться для тех же целей, что грамматика и граф переходов автомата, то есть для порождения и для распознавания предложений языка. Любой путь от входа диаграммы (соответствует начальному состоянию автомата) к её выходу (конечному состоянию *K*) порождает цепочку символов, являющуюся правильным предложением языка (сентенцией грамматики). Решение же задачи распознавания сводится к поиску такого пути от входа к выходу диаграммы, который соответствует заданной цепочке.

По сравнению с порождающей грамматикой и конечным автоматом синтаксические диаграммы гораздо нагляднее и лучше подходят для спецификации языка при его конструировании. Очень удобна диа-

грамма в роли схемы алгоритма при написании синтаксического анализатора.

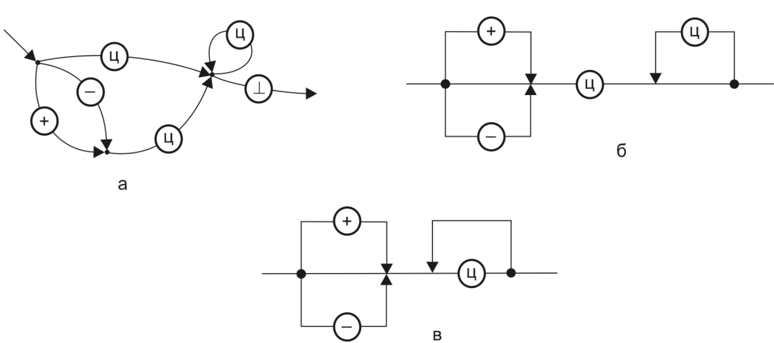


Рис. 2.16. Синтаксические диаграммы грамматики целых чисел

Используя другую манеру начертания дуг, переместив символ *ц* на дугу, ведущую в бывшее состояние *B* и отказавшись от явного изображения конца текста, получим более простую диаграмму, задающую синтаксис целых со знаком (рис. 2.16*б*). Упрощение диаграммы можно продолжить и дальше (рис. 2.16*в*), сохраняя ее эквивалентность исходной (рис. 2.16*а*). Однако, при программировании анализатора на Паскале эта последняя диаграмма не удобней предыдущей.

Перепишем анализатор целых чисел, пользуясь синтаксической диаграммой, показанной на рисунке 2.16*б*, как схемой алгоритма (листинг 2.2). Чтобы избавиться от условностей будем считать, что программа имеет доступ к глобальной переменной `Ch`, которая хранит текущий символ. Чтение следующего входного символа выполняет процедура `NextCh`, которая помещает прочитанное значение в переменную `Ch`. Считается, что константа `EOT` (от End Of Text) обозначает конец текста. Реакция на ошибку возложена на процедуру `Error`, которая выдает сообщение об ошибке и останавливает работу программы-распознавателя. В случае принятия входной цепочки никакого специального сообщения не предусматривается.

Листинг 2.2. Распознаватель целых со знаком

```
NextCh; { Прочитать первый символ }

if Ch in ['+', '-'] then
```

```
   NextCh;

if Ch in ['0'..'9'] then
   NextCh
else
   Error;

while Ch in ['0'..'9'] do
   NextCh;

if Ch <> EOT then
   Error;
```

Следует отметить ряд важных черт получившейся программы. Она состоит из нескольких частей, разделенных в листинге пустыми строками. Первая и последняя части, как нетрудно понять, должны присутствовать всегда: перед началом анализа надо получить первый символ, а по завершении — убедиться, что в момент, соответствующий выходу из диаграммы, входная цепочка исчерпана.

Три других части строго соответствуют структуре синтаксической диаграммы (см. рис. 2.16*б*). На диаграмме выделяются три последовательно соединенных участка, и программа содержит три последовательно записанных и выполняемых фрагмента. Первый из них (**`if-then`**) проверяет наличие (необязательного) знака. Второй (**`if-then-else`**) — наличие обязательной цифры. Цикл **`while`** (который, как известно, может не выполниться ни разу) соответствует циклу на диаграмме, задающему последовательность из нуля или более цифр.

Программа-распознаватель может быть написана по синтаксической диаграмме автоматной грамматики с использованием формальных приемов.

Регулярные выражения и регулярные множества

Регулярные выражения — альтернативный, отличный от порождающих грамматик и синтаксических диаграмм и имеющий свои преимущества, способ задания языка. Регулярное выражение обозначает (порождает) множество цепочек, которое называют *регулярным множеством*. Множество цепочек, соответствующее регулярному выражению R, будем обозначать R^.

Регулярные выражения над алфавитом Σ образуются по следующим правилам:

1. Отдельный символ алфавита $a \in \Sigma$ является регулярным выражением. Обозначаемое таким выражением множество цепочек есть $\{a\}$, то есть состоит из одной цепочки a.
2. Пустая цепочка ε есть регулярное выражение. Обозначает регулярное множество $\{\varepsilon\}$.
3. Если R и Q — регулярные выражения над алфавитом Σ, то запись RQ (конкатенация) также является регулярным выражением. Множество, обозначаемое RQ, состоит из всех цепочек, образованных конкатенацией двух цепочек, так, что первая цепочка пары порождается выражением R, а вторая — выражением Q. Формально это может быть записано так: $(\mathrm{RQ})\hat{} = \{\alpha\beta \mid \alpha \in R\hat{},\ \beta \in Q\hat{}\}$.
4. Если R и Q — регулярные выражения над алфавитом Σ, то запись $R \mid Q$ (читается «R или Q») также является регулярным выражением и обозначает регулярное множество $R\hat{} \cup Q\hat{}$, то есть множество всех цепочек, порождаемых как выражением R, так и выражением Q.
5. Если R — регулярное выражение над алфавитом Σ, то запись R^* (итерация R) также является регулярным выражением, и обозначает множество всех цепочек, полученных повторением цепочек, порождаемых R, ноль или более раз.
6. Если R — регулярное выражение над алфавитом Σ, то (R) (R в скобках) также является регулярным выражением, которое обозначает то же множество, что и R.

Предполагается определенный приоритет операций, с помощью которых образуются регулярные выражения. Наивысший приоритет имеет итерация (знак «*»), далее — конкатенация, далее — «или» (знак «|»). Скобки используются для изменения порядка операций.

Пример 1. С помощью регулярного выражения можно задать правила записи целых чисел со знаком:

$$(+ \mid - \mid \varepsilon)\ цц^*,$$

где $ц$ обозначает любую цифру от 0 до 9.

Если это не вызывает разночтения, символ ε можно не записывать. Повторение *один или более раз* иногда обозначают знаком «$^+$». $R^+=RR^*$. Другая форма выражения, определяющего целые:

$$(+ \mid - \mid)\ ц+.$$

Нетрудно, впрочем, записать выражение, обозначающее множество всех целых со знаком, не прибегая к условному обозначению цифр с помощью «$ц$»:

$$(+ \mid - \mid)\ (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+.$$

Пример 2. Регулярное выражение, задающее множество идентификаторов:

$$б\ (б \mid ц)^*,$$

где *б* — буква; *ц* — цифра.

Эквивалентность регулярных выражений и автоматных грамматик

Автоматные языки являются регулярными множествами. Регулярные множества являются автоматными языками.

Сказанное означает, что для любой автоматной грамматики можно записать такое регулярное выражение, что обозначаемое этим выражением множество цепочек совпадает с языком, порождаемым грамматикой. И, наоборот, для любого регулярного выражения можно найти автоматную грамматику, порождающую то же множество цепочек, что и регулярное выражение.

Будем считать, что автоматный язык задается синтаксической диаграммой. Можно установить взаимно однозначное соответствие между конструкциями, из которых строятся регулярные выражения (правила 1–6) и фрагментами, из которых состоят синтаксические диаграммы автоматных грамматик. Это соответствие показано в таблице 2.2.

Таблица 2.2. Эквивалентность регулярных выражений и автоматных грамматик

Номер правила	Фрагмент выражения	Участок диаграммы
1	*a*	a
2	ε	
3	*RQ*	R Q
4	*R* | *Q*	R Q
5	*R**	R
6	(*R*)	R

Используя такое соответствие, по выражению можно построить диаграмму, а по диаграмме — регулярное выражение. Уточнение деталей таких построений [Свердлов, 2008], которым мы не будем здесь заниматься, и доказывает справедливость сформулированного выше утверждения об эквивалентности.

Уместно напомнить, что автоматные грамматики называют также *регулярными*.

Для чего нужны регулярные выражения

Автоматные грамматики, регулярные выражения и синтаксические диаграммы являются эквивалентными способами задания автоматных языков. По сравнению с грамматиками синтаксические диаграммы обладают большей наглядностью. Регулярные же выражения имеют то важное достоинство, что представляют собой строки символов, которые могут быть легко обработаны с помощью компьютерных программ.

Обработка регулярного выражения, выступающего в роли исходных данных для некоторой программы, может иметь целью его анализ, преобразование и даже создание распознавателя автоматного языка, порождаемого этим выражением. Последнее представляет безусловный интерес, поскольку открывает возможность автоматизации построения синтаксических анализаторов. Работа подобной программы может происходить по одной из схем, показанных на рисунке 2.17.

Регулярные выражения наглядней порождающих грамматик. Это обусловлено тем, что предусмотрено явное обозначение повторения (знак итерации «*»). В нотации грамматик итерация задается с помощью рекурсии. Сравните, например, грамматики G_{10} и G_{12}, задающие язык идентификаторов, с эквивалентным регулярным выражением из примера 2.

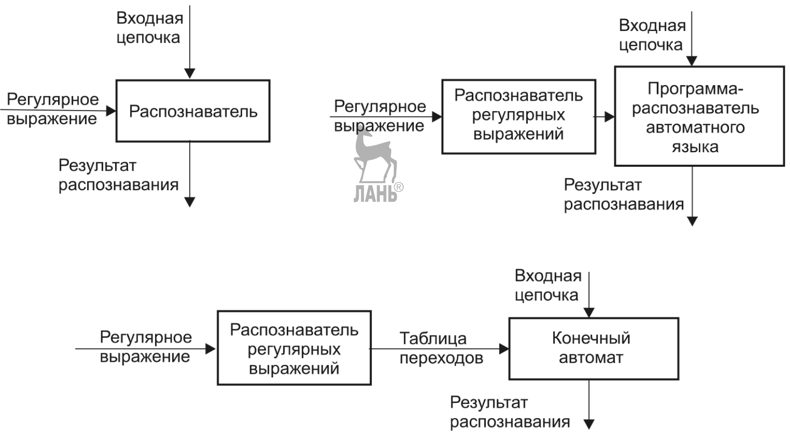


Рис. 2.17. Использование регулярных выражений

Регулярные выражения как языки

Регулярное выражение над алфавитом Σ — это цепочка символов в расширенном алфавите $\Sigma \cup \{\ |, {}^*, (,)\ \}$. Множество всех таких цепочек-выражений образует язык.

Возникает естественный вопрос, к языкам какого типа по классификации Н. Хомского этот язык принадлежит. К автоматным? Тогда, быть может, правила записи регулярных выражений можно задать регулярным выражением? Нет, нельзя. Синтаксис регулярных выражений может быть определен только контекстно-свободной, но не автоматной грамматикой. Вот эта грамматика:

$$R \to a \mid RR \mid R^* \mid R \ {''|''}\ R \mid (\ R\ ) \mid \varepsilon.$$

В этой записи есть ряд условностей: a обозначает любой символ алфавита Σ, запись ″|″, представляет знак «|», используемый в регулярных выражениях и совпадающий с аналогичным знаком, применяемым при записи грамматик.

Приведенная грамматика не отражает принятый для регулярных выражений порядок операций. Грамматика, трактующая структуру регулярного выражения в соответствии с приоритетами операций, может быть записана так:

$$R \to T \mid R \ {''|''}\ T,$$
$$T \to M \mid RM,$$
$$M \to a \mid M^* \mid (\ R\ ) \mid \varepsilon.$$

Расширенная нотация для регулярных выражений

Регулярные выражения — это строки символов, и тем они интересны как средство задания автоматных языков. Но использование надстрочных знаков «*» и «$^+$» несколько затрудняет запись выражений и их считывание компьютерной программой. Получили распространение другие варианты обозначений. Повторение ноль или более раз обозначают фигурными скобками:

$$R^* = \{R\}.$$

Используются также квадратные скобки, обозначающие необязательность заключенного в них выражения:

$$[\ R\ ] = (\ R \mid \varepsilon\ ).$$

Знаки «*» и «$^+$» в этом случае уже не используются. Соглашения о способах записи символов, с помощью которых строятся сами выражения (скобки, знак «|»), в случае, если они также входят в терминальный алфавит, могут быть разными. Можно заключать такие ме-

тасимволы в кавычки «″». При необходимости записать саму кавычку ее заключают в апострофы «′», а апостроф, если нужно, записывается в кавычках.

По этим правилам регулярные выражения, обозначающие множество целых со знаком и множество идентификаторов, будут выглядеть так:

$$[+ \mid -]\ ц\ \{ц\},$$
$$б\ \{б \mid ц\}.$$

На этом мы заканчиваем рассмотрение автоматных грамматик, в ходе которого удалось построить простые и эффективные методы распознавания автоматных языков.

С помощью автоматных грамматик определяется синтаксис простейших элементов языков программирования: идентификаторов, чисел, других констант, знаков операций и разделителей.

Контекстно-свободные (КС) грамматики и языки

К классу контекстно-свободных относятся грамматики, у которых не накладывается никаких ограничений на вид правых частей их правил, а левая часть каждого правила — единственный нетерминал. С помощью КС-грамматик задают синтаксис языков программирования.

Однозначность КС-грамматики

Как уже формулировалось выше, однозначной называется грамматика, в которой каждой сентенции соответствует единственное дерево вывода. Однако, как мы видели, дерево вывода не всегда строится явно в ходе решения задачи разбора. Поэтому имеет смысл сформулировать определение однозначности без привлечения понятия «дерево вывода».

Левосторонние и правосторонние выводы в КС-граматике

Рассмотрим (однозначную) грамматику G_8, задающую синтаксис арифметических выражений.

$$G_8\text{:}\ \begin{aligned} E &\to T \mid E + T \mid E - T \\ T &\to M \mid T * M \mid T / M \\ M &\to a \mid b \mid c \mid (\ E\ ) \end{aligned}$$

Построим два различных вывода цепочки $a + b*c$ в этой грамматике. В первом случае, если сентенциальная форма содержит более од-

ного нетерминала, будем выполнять подстановку (замену нетерминала правой частью одного из правил) для самого левого нетерминала этой сентенциальной формы:

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow M + T \Rightarrow a + T \Rightarrow a + T*M \Rightarrow a + M*M \Rightarrow$$
$$a + b*M \Rightarrow a + b*c.$$

Такой вывод называется **левосторонним**. Аналогично, вывод, в ходе которого замене всегда подвергается самый правый нетерминал сентенциальной формы, называется **правосторонним**. Для цепочки $a + b*c$ в грамматике G_8 он будет таким:

$$E \Rightarrow E + T \Rightarrow E + T*M \Rightarrow E + T*c \Rightarrow E + M*c \Rightarrow E + b*c \Rightarrow$$
$$T + b*c \Rightarrow M + b*c \Rightarrow a + b*c.$$

Нетрудно убедиться, что обе эти последовательности подстановок соответствуют одному и тому же дереву вывода, хотя и разному порядку его построения. Для цепочки $a + b*c$ в грамматике G_8 и левосторонний и правосторонний вывод строятся однозначно. Это же справедливо и для любой другой цепочки, порождаемой G_8.

КС-грамматика называется однозначной, если для каждого предложения языка, порождаемого этой грамматикой, существует единственный левосторонний вывод.

Алгоритмы распознавания КС-языков

Существует алгоритм, позволяющий для любой КС-грамматики, проверить принадлежность произвольной цепочки терминальных символов языку, порождаемому этой грамматикой, и получить вывод этой цепочки.

Наличие такого алгоритма и принцип его устройства следуют из того простого соображения, что, если имеется конечная цепочка терминалов, то для проверки ее принадлежности языку достаточно построить все возможные сентенциальные формы грамматики, имеющие длину, совпадающую с длиной этой цепочки. Количество этих сентенциальных форм конечно. Такого рода алгоритм действует по принципу полного перебора. Объем перебора может быть сокращен, если процесс порождения сентенциальных форм будет организован так, что станет возможным определить тупики в ходе перебора еще до получения сентенциальной формы нужной длины.

Переборные алгоритмы работают с возвратами и вследствие своей неэффективности мало пригодны для практического использования. Для организации перебора с возвратами используют стек — структу-

ру данных, подчиняющуюся дисциплине «последним пришел — первым ушел» (LIFO — Last In First Out).

В то же время для достаточно обширных подклассов КС-грамматик, подчиняющихся некоторым дополнительным ограничениям, существуют эффективные алгоритмы распознавания, которые мы будем в дальнейшем рассматривать и применять.

Для синтаксического анализа КС-языков используются как нисходящие (строящие дерево разбора от корня к листьям), так и восходящие алгоритмы.

Распознающий автомат для КС-языков

Для автоматных языков роль распознавателя может выполнять детерминированный конечный автомат. Существует ли универсальный автоматный распознаватель КС-языков? Да, существует.

Для произвольной КС-грамматики может быть построен недетерминированный автомат с магазинной памятью, принимающий язык, порождаемый этой грамматикой.

Автомат с магазинной памятью (магазинный автомат, МП-автомат) подобен конечному автомату, оснащенному дополнительным запоминающим устройством со стековой[53] дисциплиной «последним пришел — первым ушёл». Переходы МП-автомата определяются не только входным символом и текущим состоянием, но и значением вершины стека — элемента, поступившего в магазин последним.

Недетерминированный МП-автомат — это не что иное, как устройство, реализующее перебор с возвратами. В этом смысле он эквивалентен обсуждавшемуся выше общему алгоритму распознавания КС-языков и так же неэффективен. Для КС-грамматик, подчиняющихся определенным ограничениям, могут быть построены эффективные детерминированные магазинные автоматы, которые могут использоваться на практике для трансляции языков программирования.

Самовложение в КС-грамматиках

Если в грамматике G есть нетерминал A, для которого

$$A \overset{+}{\underset{G}{\Rightarrow}} \alpha_1 A \alpha_2,$$

[53] В литературе на русском языке стек часто называют «магазином» по аналогии с магазином автоматического оружия: патрон, заряженный в магазин последним, выстреливается первым.

то есть из A нетривиально выводится цепочка $\alpha_1 A \alpha_2$, где α_1, α_2 — непустые цепочки терминалов и нетерминалов, то говорят, что такая грамматика содержит самовложение.

Например, грамматика арифметических выражений G_8 содержит самовложение, поскольку из ее начального нетерминала E выводится цепочка (E).

$$E \Rightarrow T \Rightarrow M \Rightarrow (\ E\ ).$$

Содержит самовложение и грамматика регулярных выражений, поскольку: $R \overset{+}{\Rightarrow} (R)$.

Самовложение — характерный признак КС-грамматик.

КС-грамматика, не содержащая самовложения, эквивалентна автоматной грамматике.

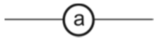
Языки арифметических и регулярных выражений являются контекстно-свободными и не могут быть заданы автоматными грамматиками.

Синтаксические диаграммы КС-языков

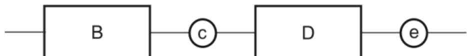
Синтаксические диаграммы КС-языка могут быть построены по его грамматике на основании следующих правил:

1. Для каждого нетерминала грамматики строится отдельная диаграмма, обозначенная названием этого нетерминала.
2. Нетерминалы из правых частей правил изображаются на диаграммах прямоугольниками, внутри которых записывается название нетерминала. Терминальные символы изображаются в кружках или овалах.
3. Для каждой правой части правила строится ветвь, представляющая собой последовательно соединенные прямоугольники и круги (овалы), следующие в том же порядке слева направо, что и соответствующие нетерминалы и терминалы правой части правила.
4. Ветви, соответствующие альтернативным правым частям правил для одного нетерминала, соединяются параллельно и образуют диаграмму для данного нетерминала.

Рассмотрим примеры построения диаграмм. Пусть в некоторой грамматике имеется правило $A \rightarrow a$, тогда на диаграмме для нетерминала A будет ветвь:

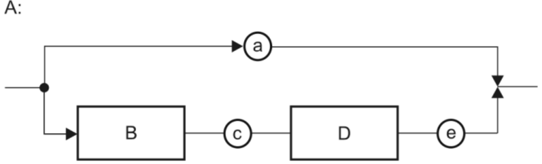


Правило $A \to BcDe$ порождает ветвь:

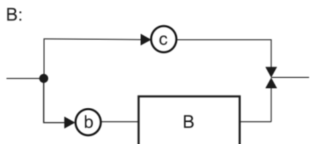


Если других правил для нетерминала A в грамматике нет, то диаграмма для этого нетерминала получается параллельным соединением ветвей. Правила для A удобней объединить в одно с альтернативными правыми частями:

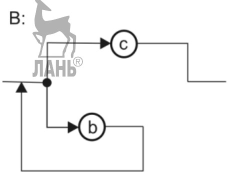
$$A \to a \mid BcDe.$$



Продолжим пример. Поскольку в правилах для A фигурируют нетерминалы B и D, то в грамматике должны быть правила, в которых B и D записаны в левой части. Пусть правило для B имеет вид: $B \to c \mid bB$. Тогда строится такая диаграмма:



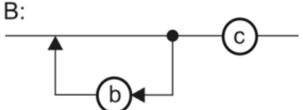
Можно, однако, заметить, что правила для B удовлетворяют ограничениям автоматных грамматик. А синтаксические диаграммы автоматных грамматик не должны содержать нетерминалов. Противоречия нет. Диаграмма для B может быть преобразована. Поскольку прохождение прямоугольного блока, обозначающего B, равносильно (порождает такую же цепочку терминалов) повторному входу в диаграмму, вход в блок B можно заменить повторным входом в диаграмму.



Такое преобразование, устранившее с диаграммы нетерминальный блок B, стало возможным благодаря тому, что нетерминал B был са-

мым правым символом в одной из альтернативных правых частей правил для B. В результате преобразования концевая (правая) рекурсия заменена циклом.

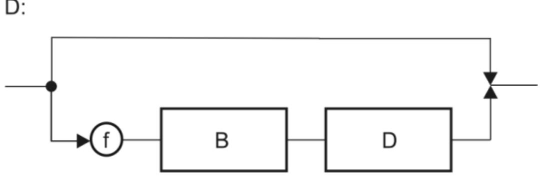
Выполнив элементарное преобразование, можно нарисовать диаграмму нетерминала B в традиционном виде:



Замена правой рекурсии циклом всегда возможна (и желательна) при построении синтаксических диаграмм КС-грамматики[54]. Интересно заметить, что сами правила 1–4 не предусматривают циклов, в то время как на практике циклы на диаграммах имеются почти всегда.

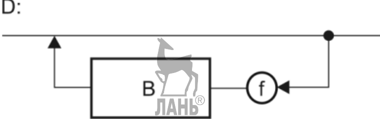
Такую же диаграмму для B можно было получить, построив фрагмент конечного автомата, а затем устранив из него состояния.

Завершая пример, зададим правило для нетерминала D: $D \rightarrow fBD \mid \varepsilon$ и построим диаграмму.



Наличие пустой цепочки в одной из альтернативных правых частей правила приводит к появлению на диаграмме параллельной ветви, в которой нет символов. Получившаяся диаграмма может быть, однако, снова упрощена:

[54] Замена концевой рекурсии циклом возможна и в программах. Если в некоторой процедуре последним выполняется рекурсивный вызов этой процедуры, то он может быть заменен переходом к началу процедуры, то есть циклом.



Определение языка с помощью синтаксических диаграмм

В действительности синтаксические диаграммы строятся, как правило, не по имеющейся грамматике, а служат самостоятельным средством проектирования языков, в том числе и языков программирования. При этом язык определяется совокупностью диаграмм, первая из которых соответствует начальному нетерминалу грамматики.

Определять синтаксис в виде совокупности диаграмм, на которых имеются нетерминальные блоки, можно не только для контекстно-свободных, но и для автоматных языков. Только из-за отсутствия самовложения диаграммы автоматного языка всегда можно объединить в одну, не содержащую нетерминалов. Для этого достаточно «подставить» в диаграмму начального нетерминала другие диаграммы вместо соответствующих прямоугольных блоков. Для КС-языка такая подстановка невозможна из-за самовложения.

Язык многочленов

Для примера построим синтаксические диаграммы, задающие правила записи (синтаксис) многочленов от x с постоянными целочисленными коэффициентами, то есть определяющие язык многочленов.

Примеры таких многочленов:

$$5x^3 + x^2 - 12x + 10,$$

$$-x,$$

$$199.$$

Последний пример может вызвать возражение, поскольку не содержит переменной x. Условимся, однако, и такую запись считать правильным многочленом нулевой степени. Чтобы запись многочленов могла быть обработана компьютерной программой (транслятором или вычислителем многочленов), предусмотрим возможность записи символов «в строку» без надстрочных показателей степени. Возведение в степень будем обозначать, как это принято в языке Бейсик и некоторых диалектах Алгола, с помощью знака «^». Тогда первый пример многочлена запишется так:

```
5x^3 + x^2 - 12x + 10
```

Построим синтаксические диаграммы, определяющие правила записи многочленов. Первой будет диаграмма для начального нетерминала, который в нашем случае есть не что иное как «Многочлен». Многочлен состоит из отдельных слагаемых, между которыми записываются знаки операций. Перед первым слагаемым также можно записать знак. Слагаемых должно быть не меньше одного. С учетом этого получается диаграмма, показанная на рисунке 2.18.

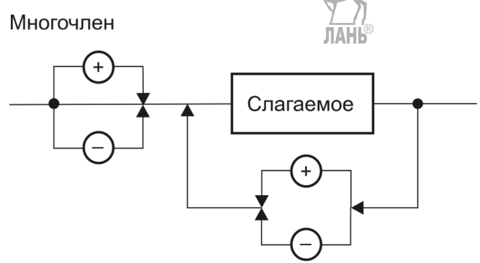


Рис. 2.18. Синтаксическая диаграмма многочлена

Теперь надо построить диаграмму для нетерминала «Слагаемое», который мы ввели в грамматику многочленов. Вначале изобразим ветвь диаграммы, соответствующую полному варианту слагаемого, когда присутствуют все его элементы: коэффициент, буква `x`, знак возведения в степень и сама степень (рис. 2.19*а*).

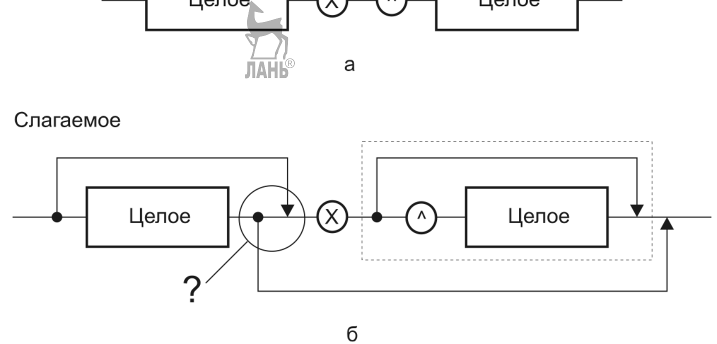


Рис. 2.19. Синтаксическая диаграмма слагаемого

Затем проведем «обходные» ветви, позволяющие предусмотреть такие варианты слагаемого, когда нет коэффициента (предполагается равным единице), буквы x и последующей степени, или только степени (см. рис. 2.19*б*). При этом не должно появиться такого пути на диаграмме, пройдя по которому мы минуем как коэффициент, так и x.

Коэффициент перед слагаемым и показатель степени записываются как целые числа без знака. Соответствующий нетерминал назван «Целое». В дальнейшем мы всегда будем считать (если не оговорено иное), что «целое» означает целое без знака. Целое без знака есть последовательность, состоящая из одной или более цифр (рис. 2.20*а*). Диаграмма для нетерминала «Цифра» показана на рисунке 2.20*б*.

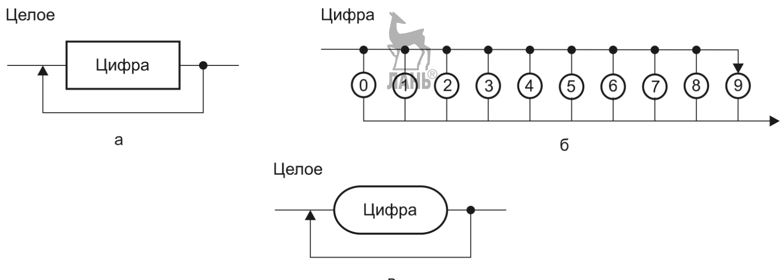


Рис. 2.20. Синтаксические диаграммы для целого и для цифры

Поскольку арабские цифры используются в самых разнообразных языках, было бы неудобно каждый раз приводить диаграмму, подобную изображенной на рисунке 2.20*б*. В дальнейшем будем вместо нетерминального блока «Цифра» использовать на диаграммах овал (рис. 2.20*в*), считая что «цифра» — это «почти терминальный символ».

Нетрудно понять (хотя бы по отсутствию самовложения), что язык многочленов — автоматный. Все диаграммы можно было бы объединить в одну, не содержащую нетерминальных блоков. Однако делать этого мы не будем. Во-первых, несколько несложных диаграмм воспринимаются проще, чем одна громоздкая. Во-вторых, использование промежуточных понятий, таких как, например, «Целое», позволяет избежать дублирования: нетерминал «Целое» встречается на диаграмме слагаемого дважды. В-третьих, представив автоматный язык с помощью КС-диаграмм, мы на простом примере рассмотрим методы распознавания КС-языков, не потеряв при этом общности подхода.

Синтаксический анализ КС-языков методом рекурсивного спуска

Рекурсивный спуск — это эффективный и простой нисходящий алгоритм распознавания. Он состоит в следующем.

Для каждого нетерминала грамматики (понятия, конструкции языка), то есть для каждой синтаксической диаграммы, записывается отдельная распознающая процедура. При этом соблюдаются следующие соглашения:

1. Перед началом работы процедуры текущим является первый символ анализируемого понятия (см. рис. 2.21).
2. В процессе работы процедура считывает все символы входной цепочки, относящиеся к данному нетерминалу (выводимые из данного нетерминала) или сообщает об ошибке. Если правила для данного нетерминала содержат в правых частях другие нетерминалы (синтаксическая диаграмма данного нетерминала содержит другие нетерминалы), то процедура обращается к распознающим процедурам этих нетерминалов для анализа соответствующих частей входной цепочки.
3. По окончании работы процедуры текущим становится первый символ, следующий во входной цепочке за данной конструкцией языка (символами, выводимыми из данного нетерминала).

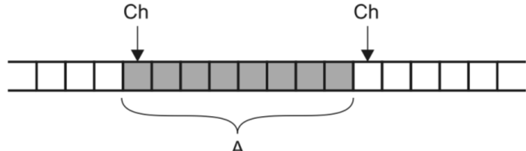


Рис. 2.21. Текущий символ в начале и конце работы распознающей процедуры нетерминала *A*

Распознавание начинается вызовом распознающей процедуры начального нетерминала. При этом текущим символом, как это следует из п. 1, должен быть первый символ входной цепочки. По завершении работы начальной процедуры текущим должен быть символ «конец текста». Таким образом, анализ методом рекурсивного спуска всегда строится по следующей схеме:

```
NextCh;{Чтение первого символа в переменную Ch}
S;     {Вызов распознающей процедуры нач. нетерминала}
if Ch <> EOT then {Проверка исчерпания входн. цепочки}
   Error;
```

Название «рекурсивный спуск» обусловлено тем, что при наличии в грамматике самовложения вызовы распознающих процедур будут рекурсивными. Процесс распознавания развивается от начального нетерминала (корень дерева разбора) через вызов процедур для промежуточных нетерминалов (внутренние вершины дерева) к анализу отдельных терминальных символов (листья дерева). Это нисходящий разбор.

Каждая распознающая процедура строится по соответствующей синтаксической диаграмме, которая играет роль схемы алгоритма. Соответствие участков диаграмм и фрагментов распознающих процедур показано в таблице 2.3. В таблице участки диаграмм обозначаются D, D_1, D_2, …, D_n. Соответствующие этим участкам фрагменты программы-распознавателя (распознающих процедур) обозначены $P(D)$, $P(D_1)$, $P(D_2)$, …, $P(D_n)$.

Таблица 2.3. Правила построения распознавателя по синтаксической диаграмме

Диаграмма D	**Распознаватель P(D)**	**Примечание**
—(a)—	`if Ch = 'a' then NextCh` `else Error;`	Анализ отдельного терминального символа
—[A]—	`A;`	Анализ нетерминала: вызов распознающей процедуры нетерминала A
— D_1 — D_2 —	`P(D1); P(D2);`	Анализ последовательно соединенных участков диаграммы

D_1
D_2
…
D_n
	`if Ch in first(D1)`
	`then`
	`   P(D1)`
	`else if Ch in first(D2)`
	`then`
	`   P(D2)`
	`…`
	`else if Ch in first(Dn)`
	`then`
	`   P(Dn)`
	`else`
		Ветвление. *first*(D_1), *first*(D_2), …, *first*(D_n) — множества направляющих символов ветвей D_1, D_2, …, D_n. Множество направляющих символов ветви D_i образуют терминальные символы, которые

Продолжение табл. 2.3

Диаграмма D	Распознаватель P(D)	Примечание
	`Error;`	могут встретиться первыми при движении по ветви D_i
	`if Ch in first(D1) then P(D1);`	Ветвление с пустой альтернативой (необязательное вхождение D_1)
	`while Ch in first(D1) do P(D1);`	Повторение D_1 ноль или более раз

Принцип работы анализатора, который строится по предлагаемым схемам, состоит в том, что, анализируя очередной символ входной цепочки, распознаватель выбирает путь движения по синтаксической диаграмме, соответствующий этой цепочке.

Пример: анализатор многочленов

Пользуясь методом рекурсивного спуска, запрограммируем синтаксический анализатор многочленов, правила записи которых определены диаграммами на рисунках 2.18–2.20. Будем считать, что анализатор должен просто отвечать на вопрос, является ли введенная пользователем строка правильно записанным многочленом.

Основная программа анализатора

Вначале необходимо подготовить текст для чтения анализатором. Было бы неправильно делать так, чтобы в основной программе и распознающих процедурах отражались особенности представления входной цепочки, то есть программировать таким образом, чтобы основные части анализатора зависели от того, считываются ли символы из файла, вводятся с терминала, извлекаются из окна редактора текста или получаются как-либо по-другому. Эта специфика будет скрыта в нескольких процедурах, которые только и будут зависеть от конкретного представления входной цепочки. Предусмотрим, что подготовка входного текста к чтению выполняется процедурой `ResetText`:

```
begin
   ResetText;
```

Далее вызываем распознающую процедуру начального нетерминала, которым в нашей задаче является «Многочлен»:

```
Polynom;
```

Процедура Polynom считывает символы входной цепочки, проверяя, соответствует ли их порядок синтаксису многочленов. Если необходимо, она обращается к другим распознающим процедурам. В случае, когда входная цепочка действительно содержит многочлен, процедура Polynom оставляет текущим (содержащимся в переменной Ch) символ, следующий за многочленом. Если при анализе многочлена обнаруживается ошибка, вызывается процедура Error, останавливающая работу всего анализатора.

Для завершения анализа остается проверить, не содержится ли за правильно записанным многочленом во входной цепочке лишних символов, то есть следует ли за многочленом символ «конец текста»:

```
   if Ch <> EOT then
      Error('Ожидается конец текста')
   else
      WriteLn('Правильно');
   WriteLn;
end.
```

Константы, переменные и вспомогательные процедуры

Теперь можно записать начало программы с описаниями констант и переменных, использованными в основном блоке. Кроме глобальной переменной Ch, обозначающей текущий символ, предусмотрим глобальную переменную Pos, которая будет хранить номер этого символа во входной цепочке.

```
program ParsePoly;
const
   EOT = chr(0);     { Признак "конец текста" }
var
   Ch  : char;       { Очередной символ    }
   Pos : integer;    { Номер символа       }
```

Взаимодействие с входным текстом будут выполнять процедуры ResetText — готовит входную цепочку к считыванию распознавателем, и NextCh — читает очередной символ входной цепочки, помещая его в переменную Ch.

Использование глобальных переменных в процедурах, вообще-то, плохая практика. Хотя мы еще не начали обращаться к Ch и Pos

в процедурах распознавателя, но вот-вот сделаем это. Оправданием служит специфика задачи. Синтаксический анализатор и компилятор — своеобразные программы, в которых используется не слишком много переменных. А в нашем анализаторе переменных будет и вовсе две — только что определенные `Ch` и `Pos`. Их передача в процедуры через параметры загромоздила бы программу, в то время как обращение за очередным символом, например, к процедуре `NextCh` все равно выглядело бы всегда одинаково: `NextCh(Ch, Pos)`.

Предусмотрим чтение исходных данных (записи многочлена) из стандартного входного файла (по умолчанию — ввод с клавиатуры). Сообщения распознавателя будут выводиться в стандартный выходной файл (т. е. на экран). В этом случае процедура, подготавливающая текст, запишется так:

```
{ Подготовить текст }
procedure ResetText;
begin
   WriteLn('Введите многочлен от x с целыми коэфф-
тами');
   Pos := 0;
   NextCh; { Чтение первого символа }
end;
```

При чтении очередного символа будем игнорировать пробелы, считая их незначащими. Это позволит при записи исходного многочлена применять пробелы, например, для отделения одного слагаемого от другого.

```
procedure NextCh;
{ Читать следующий символ }
begin
   repeat
      Pos := Pos+1;
      if not eoln then
         Read(Ch)
      else begin
         ReadLn;
         Ch := EOT;
      end;
   until Ch <> ' ';
end;
```

Такой пропуск пробелов делает их допустимыми в любом месте записи многочлена. К примеру, такая строка должна теперь считаться правильной:

```
1 2 3 x ^ 4 + 5 6 7 x + 8 9
```

Это не соответствует соглашениям современных языков программирования. Более совершенное решение вопроса о пробелах будет рассмотрено в следующей главе, пока же примем простейший подход, который все же предпочтительней полного запрета пробелов.

Процедура, реагирующая на ошибку, тоже зависит от соглашений по вводу и выводу. Как мы уже условились, она будет выдавать сообщение на экран:

```
procedure Error(Message: string); { Ошибка  }
   { Message - сообщение об ошибке }
begin
   WriteLn('^': Pos);
   WriteLn('Синтаксическая ошибка: ', Message);
   Halt; { Прекращение работы анализатора }
end;
```

Вывод знака «'^'» в позиции `Pos` позволяет указать стрелкой на символ, вызвавший ошибку. Диалог с анализатором может быть таким:

```
Введите многочлен от X с целыми коэфф-тами
2x + 1.2
      ^

Синтаксическая ошибка: Ожидается конец текста
```

Реакция программы в этом примере может показаться непонятной, хотя она совершенно корректна. Получение такого сообщения означает: «Если бы на этом месте запись многочлена закончилась, было бы синтаксически правильно. Но в указанном месте стоит неподходящий символ». Понятно, что распознаватель не может знать наших желаний и содержательно реагировать на попытку записать вещественное число вместо целого.

Распознающие процедуры

Для каждого нетерминала грамматики многочленов, то есть для каждой синтаксической диаграммы (рис. 2.18–2.20) записываем одну распознающую процедуру. Первой — процедуру для нетерминала «Многочлен» — начального нетерминала грамматики (рис. 2.18). Трудность, однако, состоит в том, что диаграмма на рисунке 2.18 не-

удобна для программирования — она содержит цикл с выходом из середины, в то время как в Паскале такого цикла нет. Заменим диаграмму эквивалентной, содержащей цикл с предусловием (с выходом в начале).

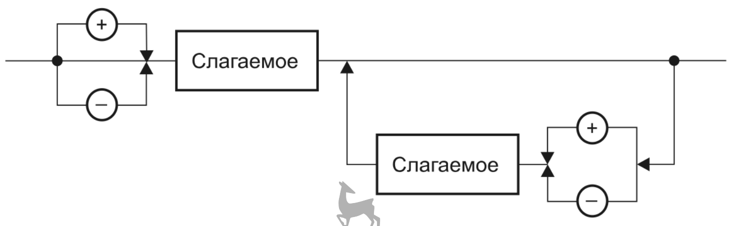


Рис. 2.22. Преобразованная диаграмма «Многочлен»

Теперь не составляет труда записать распознающую процедуру, структура которой в точности повторяет структуру диаграммы: на диаграмме три последовательно соединенных участка — в процедуре три оператора, выполняемых один за другим: **if**, вызов процедуры Addend, цикл **while**.

```
procedure Polynom; { Многочлен }
begin
   if Ch in ['+', '-'] then
      NextCh;
   Addend; { Слагаемое }
   while Ch in ['+', '-'] do begin
      NextCh;
      Addend;
   end;
end;
```

Следующий нетерминал — «Слагаемое». Однако диаграмма, показанная на рисунке 2.19, не разделяется на типовые фрагменты, что затрудняет программирование распознавателя. Неструктурированность обусловлена фрагментом, который помечен на рисунке знаком «?». Преобразуем диаграмму в эквивалентную, но состоящую только из совокупности типовых структур (рис. 2.23). Для этого изобразим отдельно две ветви: одна соответствует слагаемому, начинающемуся с числа, другая — с буквы *x*. Фрагмент, выделенный на исходной диаграмме пунктирной рамкой, преобразуем в нетерминал «Степень».

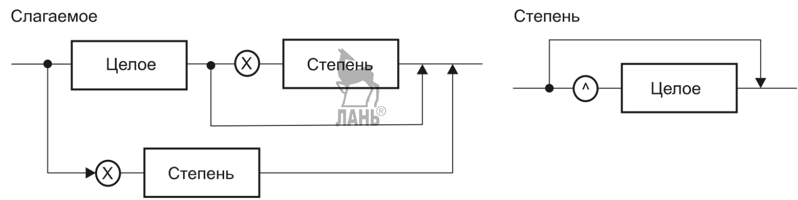


Рис. 2.23. Синтаксические диаграммы «Слагаемое» и «Степень»

Программирование распознающих процедур `Addend` (слагаемое) и `Power` (степень) теперь выполняется легко: диаграммы служат схемами алгоритмов.

```
procedure Addend; { Слагаемое }
begin
   if Ch = 'x' then begin
      NextCh;
      Power;  { Степень }
      end
   else begin
      Number; { Целое }
      if Ch = 'x' then begin
         NextCh;
         Power;
      end;
   end;
end;

procedure Power; { Степень }
begin
   if Ch = '^' then begin
      NextCh;
      Number;
   end;
end;
```

Полезно обратить внимание на дисциплину вызова процедуры `NextCh`. Следующий символ считывается, когда опознан текущий.

Последняя распознающая процедура — для «Целого». И в этом случае тоже можно преобразовать исходную диаграмму (см. рис. 2.20), отделив блок, соответствующий первой цифре (обязательной), от остальных блоков (рис. 2.24).

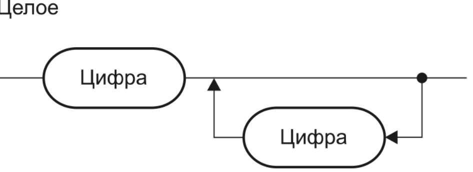


Рис. 2.24. Синтаксическая диаграмма «Целое»

Предусматривать отдельную распознающую процедуру для «почти терминального символа» «цифра» неразумно. Проще и наглядней непосредственно проверять принадлежность очередного знака множеству цифр.

```
procedure Number; { Целое }
begin
   if Ch in ['0'..'9'] then
      NextCh
   else
      Error('Число начинается не с цифры');
   while Ch in ['0'..'9'] do
      NextCh;
end;
```

Распознаватель готов. Осталось только расположить в программе написанные процедуры в правильном порядке — описание процедуры поместить перед ее вызовом. Поскольку рекурсии в этом примере нет, сделать это легко.

На рассмотренном примере мы продемонстрировали, что синтаксический анализатор методом рекурсивного спуска можно написать «почти так же быстро, как мы вообще можем писать» [Хантер, 1984].

Требование детерминированного распознавания

Уже в ходе предыдущего рассмотрения можно было заметить, что рекурсивный спуск позволяет построить анализатор не для любой КС-грамматики. Ограничения возникают при анализе направляющих символов отдельных ветвей синтаксической диаграммы.

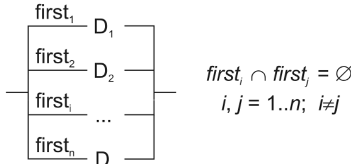
Множество направляющих символов first(D) ветви D синтаксической диаграммы образуют терминальные символы, которые могут встретиться первыми при движении по диаграмме вдоль этой ветви.

Движение по диаграмме предполагает, что при достижении прямоугольного блока, изображающего нетерминал, оно продолжается по

диаграмме этого нетерминала с последующим возвратом на исходную диаграмму.

Порядок действий анализатора, работающего по методу рекурсивного спуска, и определяется анализом направляющих символов отдельных ветвей синтаксической диаграммы. Чтобы алгоритм работал без возвратов, выбор направления движения по диаграмме выполнялся однозначно, должно соблюдаться требование детерминированного распознавания:

В каждом разветвлении синтаксической диаграммы множества направляющих символов отдельных ветвей не должны попарно пересекаться.



В рассмотренной выше грамматике многочленов требование детерминированного распознавания всегда соблюдается.

Возьмем, например, диаграмму, показанную на рисунке 2.22. На ней две точки ветвления. Первая — на входе в диаграмму, вторая — на выходе. В первом разветвлении три ветви. Множества направляющих символов этих ветвей: `['+']` — верхняя ветвь; `['0'..'9', 'X']` — средняя ветвь; `['-']` — нижняя ветвь. Как видно, эти множества не пересекаются.

В правой точке происходит ветвление на два направления. Одно ведет на выход из диаграммы, множество направляющих символов этой ветви состоит из символа «конец текста»: `[⊥]`. Другая ветвь соответствует очередному витку цикла, ее множество направляющих символов равно `['+', '-']`. Пересечения множеств снова нет.

Нетрудно убедиться, что не пересекаются и множества направляющих символов отдельных ветвей диаграмм, показанных на рисунках 2.23 и 2.24.

LL-грамматики

LL(*k*)*-грамматикой называется КС-грамматика, в которой выбор правила в ходе левостороннего вывода однозначно определяется не*

более чем k очередными символами входной цепочки, считываемой слева направо.

Название «*LL*» происходит от двух слов «left» (левый), встречающихся в описании хода распознавания в *LL*-грамматике: левосторонний вывод при чтении слева. Самыми удобными для распознавания, конечно же, являются *LL*(1) грамматики, в которых выбор направления распознавания однозначно определяется очередным входным символом.

Сформулированное выше требование детерминированного распознавания при рекурсивном спуске есть не что иное, как необходимость того, чтобы используемая грамматика относилась к классу *LL*(1).

Рекурсивный спуск — это детерминированный метод нисходящего разбора КС-языков, порождаемых LL(1)-грамматиками.

Левая и правая рекурсия

Рассмотрение грамматик с левой и правой рекурсией позволит нам получить некоторые признаки, помогающие определить наличие или отсутствие *LL*(1) свойства у КС-грамматик.

Если в грамматике *G* существует нетерминал *A*, для которого $A \overset{+}{\underset{G}{\Rightarrow}} A\alpha$, где α — непустая цепочка, грамматика содержит левую рекурсию.

Грамматика, содержащая левую рекурсию, не может быть LL(1) грамматикой.

Если в грамматике *G* существует нетерминал *A*, для которого $A \overset{+}{\underset{G}{\Rightarrow}} \alpha A$, где α — непустая цепочка, грамматика содержит правую рекурсию.

Леворекурсивная грамматика всегда может быть преобразована в эквивалентную праворекурсивную.

Синтаксический анализ арифметических выражений

Рассмотрим арифметические выражения, синтаксис которых задается грамматикой:

$$G_8: \quad E \to T \mid E + T \mid E - T,$$
$$T \to M \mid T * M \mid T / M,$$
$$M \to a \mid b \mid c \mid (E).$$

Эта КС-грамматика, построенная нами раньше, обладает рядом достоинств. Она однозначна и ассоциирует операнды в соответствии с общепринятым порядком выполнения операций, когда вначале выполняются умножение и деление, затем — сложение и вычитание. Однако нетрудно видеть, что G_8 леворекурсивна. Действительно, для нетерминала E справедливо: $E \Rightarrow E + T$. Наличие левой рекурсии препятствует использованию рекурсивного спуска.

G_8 может быть заменена эквивалентной (порождающей тот же язык) праворекурсивной грамматикой:

$$
\begin{aligned}
G_{13}:\quad & E \to T \mid T + E \mid T - E,\\
& T \to M \mid M * T \mid M / T,\\
& M \to a \mid b \mid c \mid (\, E\,).
\end{aligned}
$$

Хотя левой рекурсии больше нет, грамматика G_{13} не является $LL(1)$-грамматикой. Чтобы убедиться в этом, достаточно обратиться к правилам для нетерминала E:

$$E \to T \mid T + E \mid T - E.$$

Напомню, что эту строку следует рассматривать как сокращенную запись трех альтернативных правил для нетерминала E. Очевидно, что выбор одного из этих трех правил на основе анализа одного входного символа невозможен, поскольку правая часть каждого правила начинается одним и тем же нетерминалом T. На рисунке 2.25 показана синтаксическая диаграмма нетерминала E грамматики G_{13}.

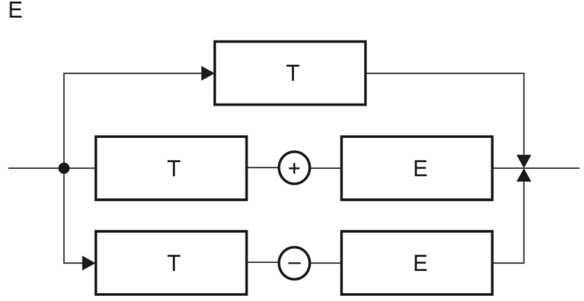


Рис. 2.25. Диаграмма нетерминала E грамматики G_{13}

Рассмотрение диаграммы убеждает, что требование детерминированного распознавания не выполнено: множества направляющих символов всех трех ветвей совпадают: $first_1 = first_2 = first_3 = \{\, a,\ b,\ c,\ (\,\}$.

Грамматика G_{13} и соответствующие синтаксические диаграммы могут быть преобразованы так, чтобы использование рекурсивного спуска стало возможным. Особенно легко увидеть возможность преобразования диаграмм. Действительно, достаточно вынести блок *T* из трех параллельных ветвей и поместить его в общую ветвь, как требование детерминированного распознавания для диаграммы нетерминала *E* будет соблюдено (рис. 2.26).

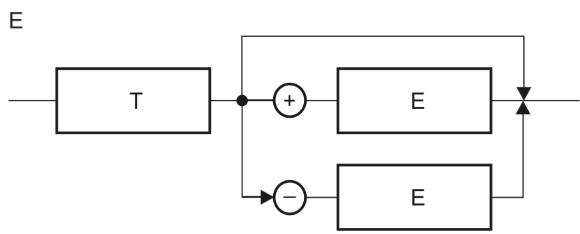


Рис. 2.26. Преобразованная диаграмма нетерминала *E* грамматики выражений

Аналогичное изменение может быть выполнено и для диаграммы слагаемого (нетерминал *T*).

Менее очевиден способ преобразования грамматики. Потребуется два дополнительных нетерминала. Обозначим их *A* и *B*. Тогда новая грамматика может быть записана так:

$$
\begin{aligned}
G_{14}:\quad & E \to T\,A, \\
& A \to \varepsilon \mid +E \mid -E, \\
& T \to M\,B, \\
& B \to \varepsilon \mid *\,T \mid /\,T, \\
& M \to a \mid b \mid c \mid (\,E\,).
\end{aligned}
$$

Содержательно *A* и *B* можно трактовать как «выражение без первого слагаемого» и «слагаемого без первого множителя» соответственно.

G_{14} — это *LL*(1)-грамматика для арифметических выражений. Но и она не вполне может нас устроить. Дело в том, что G_{14}, как и G_{13} (но не G_8), связывает операнды нескольких идущих подряд операций одного приоритета неподходящим образом, группируя их справа налево. На рисунке 2.27*а* показано дерево вывода выражения $a - b - c$ в грамматике G_{14}.

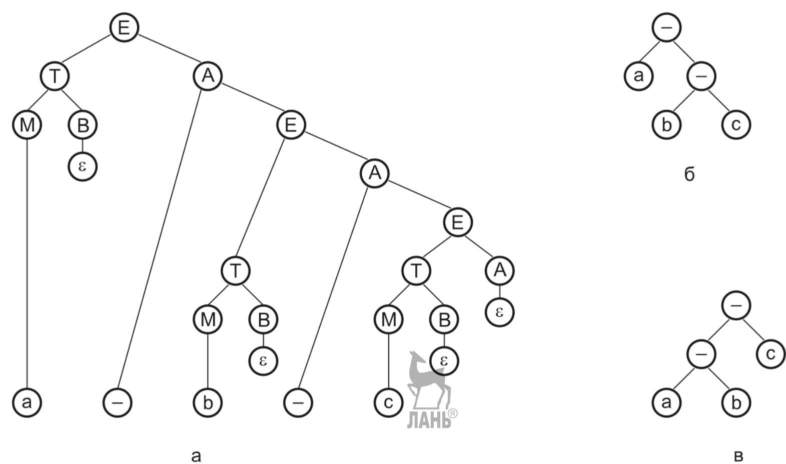


Рис. 2.27. Дерево выражения $a - b - c$

Устранив из этого синтаксического дерева нетерминалы (и ε) и перенося знаки операций во внутренние вершины, получим семантическое дерево выражения (рис. 2.27*б*), которое, увы, не соответствует правильному порядку его вычисления: оно подразумевает группировку операндов $a - (b - c)$, в то время как $a - b - c = (a - b) - c$. Правильное семантическое дерево можно видеть на рисунке 2.27*в*.

Модернизируем G_{14} с целью обеспечить правильную ассоциацию операций и операндов. Получаем грамматику

$$
\begin{aligned}
G_{15}:\quad & E \to T\,A, \\
& A \to \varepsilon \mid +\,TA \mid -\,TA, \\
& T \to M\,B, \\
& B \to \varepsilon \mid *\,MB \mid /\,MB, \\
& M \to a \mid b \mid c \mid (\,E\,).
\end{aligned}
$$

Эта однозначная праворекурсивная $LL(1)$-грамматика приписывает выражению правильную структуру. По ней без труда может быть написан синтаксический анализатор, работающий по алгоритму рекурсивного спуска. Интересно отметить, что программа-распознаватель, написанная по этой грамматике, будет рекурсивной (что, как известно, эквивалентно итерации), но не будет содержать циклов. Недостатками такой грамматики является некоторая ее громоздкость и наличие нетерминалов A и B с неочевидным смыслом.

Удовольствие написать распознаватель по грамматике G_{15} предоставлю читателям. Запрограммировать его вы сможете так же быстро, как быстро умеете писать.

А теперь мы воспользуемся теми выразительными средствами, которые дают синтаксические диаграммы. Это в первую очередь возможность использования цикла вместо рекурсии. Вообще, можно заметить, что некоторые трудности, возникающие на практике при использовании грамматик Хомского, обусловлены тем, что с их помощью приходится выражать повторение через рекурсию. Так, о выражении, представляющем собой просто последовательность слагаемых, мы вынуждены думать как о рекурсивной конструкции: выражение — это первое слагаемое, за которым после знака снова записано выражение (правая рекурсия, неподходящая группировка слагаемых: первое слагаемое складывается со всем остальным выражением). Или: если к выражению прибавить или от выражения отнять слагаемое, то снова получится выражение (левая рекурсия, правильная группировка слагаемых). Эти проблемы могут быть устранены добавлением в нотацию формальных грамматик явного обозначения для повторения, что и будет сделано при рассмотрении грамматик языков программирования. Синтаксические диаграммы также позволяют обойтись без рекурсии там, где она служит всего лишь для задания повторений.

Рассмотрим диаграмму нетерминала *E* (см. рис. 2.26). Налицо рекурсия: на диаграмме *E* присутствует нетерминальный блок *E*. Но это правая, концевая рекурсия. Обращение к блоку *E* можно заменить переходом на вход диаграммы (рис. 2.28*а*).

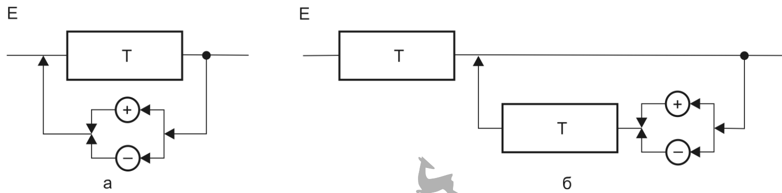


Рис. 2.28. Синтаксическая диаграмма выражения, не содержащая рекурсии

Такую диаграмму для выражения можно было построить и без выписывания и преобразования грамматики. Суть изображенного проста: выражение — это последовательность одного или более слагае-

мых, между которыми записываются знаки «+» или «–». Поскольку обработка слагаемых в ходе синтаксического анализа будет происходить последовательно в цикле, не составит труда организовать трансляцию так, чтобы это соответствовало выполнению операций слева направо.

Чтобы программировать распознаватель на Паскале было удобней, преобразуем диаграмму в эквивалентную, содержащую цикл с предусловием (рис. 2.28*б*). Аналогично строится диаграмма слагаемого (нетерминал *T*) (рис. 2.29*а*). Диаграмма множителя соответствует правилам для нетерминала *M* в грамматиках G_8, G_{13}, G_{14}, G_{15} (рис. 2.29*б*).

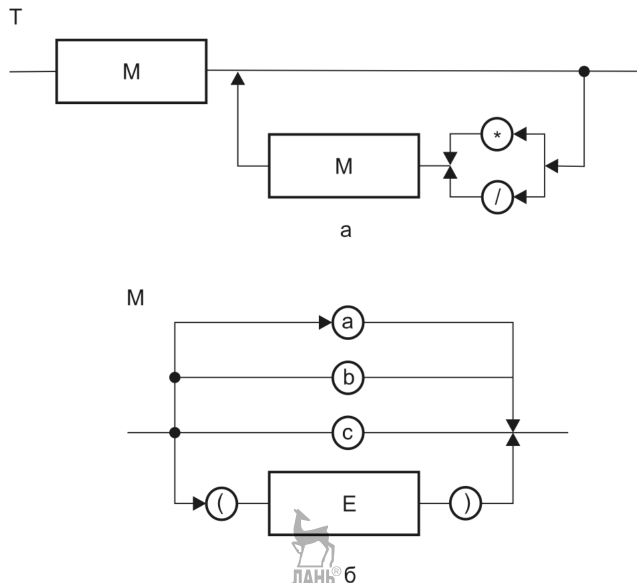


Рис. 2.29. Синтаксические диаграммы слагаемого и множителя

Теперь можно программировать анализатор. Запишем только распознающие процедуры. Константы, переменные и вспомогательные процедуры предполагаются такими же, как в распознавателе многочленов.

Начать естественно с процедуры, соответствующей начальному нетерминалу, то есть с процедуры *E*:

```
procedure E;
begin
   T;
   while Ch in ['+', '-'] do begin
      NextCh;
      T;
   end;
end;
```

Далее записываем распознаватель слагаемого:

```
procedure T;
begin
   M;
   while Ch in ['*', '/'] do begin
      NextCh;
      M;
   end;
end;
```

И, наконец, распознающую процедуру для нетерминала *M* — множителя:

```
procedure M;
begin
   if Ch in ['a', 'b', 'c'] then
      NextCh
   else if Ch = '(' then begin
      NextCh;
      E;
      if Ch = ')' then
         NextCh
      else
         Error('Ожидается ")"');
      end
   else
      Error('Ожидается a, b, c или "("');
end;
```

Остается только один вопрос: как разместить эти три процедуры в программе? По общему правилу языка Паскаль любой идентификатор может быть использован только после его описания. Процедура `E` вызывает процедуру `T`, поэтому должна располагаться после нее. Процедура `T` вызывает `M`, поэтому `M` надо разместить перед `T`. Получается такой порядок: `M`; `T`; `E`. Но из процедуры `M` вызывается `E`, в то время как описание `E` мы разместили после `M`. Выходом из этой ситуа-

ции, которая возникла из-за наличия в нашей программе косвенной рекурсии, является использование директивы **forward** для опережающего описания процедуры E. Структура программы будет такой:

```
procedure E; forward;
procedure M;
…
procedure T;
…
procedure E;
…
```

Есть и другое, весьма изящное решение проблемы: использование вложенных процедур. Такой распознаватель представлен в листинге 2.3.

Листинг 2.3. Распознаватель арифметических выражений

```
procedure E;

   procedure T;

      procedure M;
      begin
         if Ch in ['a', 'b', 'c'] then
            NextCh
         else if Ch = '(' then begin
            NextCh;
            E;
            if Ch = ')' then
               NextCh
            else
               Error('Ожидается ")"');
            end
         else
            Error('Ожидается a, b, c или "("');
      end {M};

   begin
      M;
      while Ch in ['*', '/'] do begin
         NextCh;
         M;
      end;
   end {T};

begin
```

```
    T;
    while Ch in ['+', '-'] do begin
       NextCh;
       T;
    end;
 end {E};
```

ЛАНЬ®

Включение действий в синтаксис

Синтаксический анализатор отвечает на вопрос о принадлежности входной цепочки языку. В ходе распознавания выявляется структура входного текста. Эта структура может быть представлена явно, например, в виде дерева, или неявно — последовательностью действий, совершенных распознавателем.

На основе распознавания структуры входного текста строится и его содержательная обработка, трансляция. Синтаксический анализатор служит основой, остовом транслятора, предоставляя возможность выполнить необходимые действия по смысловой (семантической) обработке в нужные моменты в соответствии со структурой входной цепочки.

Семантические процедуры

Встраиваемые в распознаватель действия, предназначенные для выполнения смысловой обработки входного текста, будем называть *семантическими процедурами*.

Слово «процедура» употребляется здесь в широком смысле, как определённая последовательность действий. В программе-распознавателе это может быть не обязательно процедура-подпрограмма, но и просто один или несколько операторов.

Рассмотрим использование семантических процедур на простом примере.

ЛАНЬ®

Пусть речь идет о распознавателе целых чисел без знака, который был использован в программе анализаторе многочленов. Теперь в его задачу будет входить получение значения числа, представленного последовательностью цифр.

Обозначив семантические процедуры P_1 и P_2, разместим на синтаксической диаграмме «Целое» (рис. 2.30) соответствующие им треугольные значки в тех местах, при прохождении которых во время анализа эти процедуры должны выполняться.

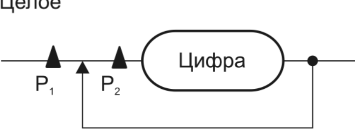


Рис. 2.30. Семантические процедуры на диаграмме целого без знака

Процедура P_1 будет выполняться в начале обработки и присвоит искомому числу исходное нулевое значение. Переменная `y` — это формируемое значение числа.

```
P1: y := 0;
```

Задача P_2 — добавлять к числу справа очередную цифру. Для этого «старое» значение `y` умножим на 10 (основание десятичной системы счисления) и добавим значение прочитанной цифры. Условимся, что семантическая процедура, значок которой размещен перед обозначением терминального символа, выполняется в тот момент, когда этот символ является текущим (содержится в переменной `Ch`). Надо побеспокоиться и о том, чтобы при вычислениях не произошло переполнения — то есть не получилось число, превышающее максимально допустимое целое.

```
P2: d := ord(Ch) - ord('0');
    if y <= (maxint - d) div 10 then
       y := 10*y + d
    else
       Error('Слишком большое число');
```

Вы, конечно, понимаете, что выполнять такую проверку

```
if 10*y + d <= maxint then …
```

было бы, как минимум, наивно — переполнение случится еще до того, как дело дойдет до сравнения.

Теперь можно написать распознающую процедуру, выполняющую не только синтаксический анализ, но и получение числового значения целого, которое будет выходным параметром этой процедуры.

```
procedure Number(var y : integer); { Целое }
var
   d : integer;
begin
   y := 0;                                    { P1 }
   if not( Ch in ['0'..'9'] ) then
```

```
      Error('Число начинается не с цифры');
   repeat
      d := ord(Ch) - ord('0');                    { P2 }
      if y <= (maxint - d) div 10 then            { P2 }
         y := 10*y + d                            { P2 }
      else                                        { P2 }
         Error('Слишком большое число');          { P2 }
      NextCh;
   until not( Ch in ['0'..'9'] );
end;
```

Напомню, что процедура `Error` останавливает работу всего анализатора, поэтому не надо беспокоиться, что при возникновении ошибки в цикле работа этого цикла разладится. В дальнейшем мы обсудим другие варианты обработки ошибок, но пока такое соглашение позволяет не загромождать транслятор дополнительными проверками. Можно еще отметить, что ошибка «Слишком большое число» уже не синтаксическая. Она связана с содержательной обработкой, и вправе называться семантической.

Пример: умножение многочленов

Рассмотрим задачу, которая была предложена на одной из олимпиад по программированию.

ЗАДАЧА

> Составить программу, вводящую в символьной форме два многочлена от x с целыми коэффициентами и выводящую их произведение в символьной форме в порядке убывания степеней. Суммарная степень многочленов не превышает 255.

Вот пример работы такой программы:

```
Перемножение многочленов
------------------------
1-й многочлен:
3x^2+3x-5
2-й многочлен:
x^3 - 2x
Произведение равно:
3x^5 + 3x^4 - 11x^3 - 6x^2 + 10x
```

Программа печатает запрос, в ответ на который можно ввести запись первого, а затем второго многочлена в привычной, использую-

мой в математике форме. Обработав полученные данные, программа печатает результат — произведение многочленов в таком же естественном виде.

Проектирование

Приступим к решению поставленной задачи. Сформулируем общий план:

1. Напечатать заголовок.
2. Ввести текст 1-го многочлена, проанализировать его запись и преобразовать в удобную для дальнейших вычислений форму.
3. Для выполнения действий с многочленами (в том числе перемножения) удобно хранить их в программе в виде массива коэффициентов, количество которых соответствует степени многочлена, а индекс каждого коэффициента равен степени соответствующего слагаемого.
4. Ввести текст 2-го многочлена, проанализировать его запись и преобразовать в удобную для дальнейших вычислений форму.
5. Перемножить многочлены.
6. Имея представление обоих многочленов-сомножителей во внутреннем формате и выполняя необходимые вычисления с их коэффициентами, получаем коэффициенты многочлена-произведения.
7. Напечатать результат.

Запишем начало программы, в котором в соответствии с уже принятыми решениями определим необходимые константы, типы данных и переменные.

```
program MultPoly; { Перемножение многочленов }
const
   nmax = 255;          { Максимальная степень }
type
   tPoly = record { Тип многочленов }
      n: integer;                    { Степень }
      a: array [0..Nmax] of integer; { Коэффициенты }
   end;
var
   P1, P2, Q: tPoly;    { Сомножители и произведение }
```

Обратите внимание, что многочлены отнесены к типу tPoly, который представляет собой запись, содержащую, кроме упоминавшегося массива коэффициентов, величину n — фактическую степень.

Теперь, пользуясь предварительно составленным планом, запишем основную программу.

```
begin
   WriteLn('Перемножение многочленов');
   WriteLn('------------------------');
   WriteLn;
   WriteLn('1-й многочлен');
   GetPoly(P1);
   WriteLn('2-й многочлен');
   GetPoly(P2);
   { Перемножение }
      MultPoly(P1, P2, Q);
   WriteLn;
   WriteLn('Произведение:');
   { Печать результата }
      WritePoly(Q);
   WriteLn;
end.
```

Умножение и вывод

Детализацию начнем с процедуры, которая выполняет перемножение. Её входными параметрами являются многочлены-сомножители, выходным — многочлен-произведение. И входные, и выходной параметры являются многочленами, представленными в виде совокупности массива коэффициентов и величины, задающей степень многочлена, то есть относятся к типу `tPoly`. Обозначив сомножители `X` и `Y`, а результат — `Z`, запишем заголовок процедуры.

```
procedure MultPoly(X, Y: tPoly; var Z: tPoly);
```

Основная идея вычисления коэффициентов многочлена-произведения состоит в том, что при попарном перемножении слагаемых первого и второго многочлена получающееся произведение участвует (в качестве одного из слагаемых) в формировании того слагаемого многочлена-результата, степень которого равна сумме степеней сомножителей. То есть при перемножении i-го члена многочлена `X` и j-го члена многочлена `Y` получается величина, которая должна быть добавлена к $i+j$-му слагаемому `Z`:

```
Z.a[i+j] := Z.a[i+j] + X.a[i]*Y.a[j];
```

Такое вычисление нужно выполнить для всех сочетаний i и j, не забыв присвоить нулевые начальные значения коэффициентам `Z` и побеспокоиться об определении степени многочлена `Z`. Получаем:

```
{ Умножение многочленов. Z = X*Y }
procedure MultPoly(X, Y: tPoly; var Z: tPoly);
```

```
var
   i, j: integer;
begin
   ClearPoly(Z);    { "Обнуление" Z }
   for i := 0 to X.n do
      for j := 0 to Y.n do
         Z.a[i+j] := Z.a[i+j] + X.a[i]*Y.a[j];
   { Определение степени многочлена Z }
      Z.n := nmax;
      while ( Z.n>0 ) and ( Z.a[Z.n]=0 ) do
         Z.n := Z.n-1;
end;
```

Процедура `ClearPoly`, выполняющая «обнуление» многочлена, выглядит так:

```
procedure ClearPoly(var P : tPoly);
var
   i : integer;
begin
   for i := 0 to nmax do
      P.a[i] := 0;
   P.n := 0;
end;
```

Теперь займемся печатью многочлена. Слагаемые должны выводиться в порядке убывания степеней. Слагаемому может предшествовать знак. Коэффициент (при ненулевой степени) печатается, если он не равен 0 или 1. Буква `x` выводится для ненулевых степеней, а значение показателя степени (и знак "^" перед ним) — если эта степень больше единицы.

```
{ Вывод многочлена }
procedure WritePoly(P : tPoly);
var
   i : integer;
begin
   with P do
      for i := n downto 0 do begin
         if ( a[i]>0 ) and ( i<>n ) then
            Write(' + ')
         else if ( a[i]<0 ) and ( i=n ) then
            Write('-')
         else if a[i]<0 then
            Write(' - ');
         if   ( abs(a[i])>1 ) or
```

```
            ( i=0 ) and ( a[i] <> 0 ) or
            ( n=0 )
         then
            Write(abs(a[i]));
         if ( i>0 ) and ( a[i]<>0 ) then
            Write('x');
         if ( i>1 ) and ( a[i]<>0 ) then
            Write('^', i)
      end;
end;
```

Транслятор многочленов

Нам осталось реализовать транслятор (процедуру `GetPoly`), преобразующий введенную запись многочлена в массив коэффициентов. Его задача — считывать символы из входной строки, выполнять распознавание многочлена и вычислять его коэффициенты. Распознающие процедуры будут вложены внутрь `GetPoly`, а сама эта процедура лишь подготовит чтение входной строки, вызовет распознаватель и убедится, что за многочленом во входной строке ничего не содержится.

```
{ Ввод и трансляция многочлена }
procedure GetPoly(var P: tPoly);
const
   EOT = chr(0);      { Конец текста }
var
   Pos: integer;   { номер символа }
   Ch : char;      { очередной символ }
…
{ Здесь разместятся процедуры распознавателя }
…
begin
   Pos := 0;
   NextCh;
   Poly(P);
   if Ch <> EOT then
      Error('Ожидается конец текста');
end;
```

Разместим на синтаксических диаграммах значки семантических процедур и определим содержание этих процедур. Вначале на диаграмме многочлена (рис. 2.31).

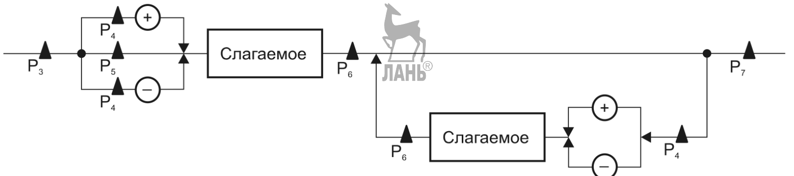


Рис. 2.31. Синтаксическая диаграмма многочлена с семантическими процедурами

Перед началом обработки массив коэффициентов многочлена заполняется нулями, а его степень принимается равной нулю. Эту работу выполнит семантическая процедура P_3, вызвав уже написанную нами процедуру `ClearPoly`.

```
P3: ClearPoly(P);
```

P_4 и P_5 отвечают за запоминание знака перед слагаемым. Знак сохраняется в локальной переменной `Op`.

```
P4: Op := Ch;
P5: Op := '+';
```

Каждое слагаемое многочлена имеет вид $a_k x^k$. Транслятор слагаемого вычислит значения коэффициента a и степени k. Получив эти значения, семантическая процедура P_6 в зависимости от знака добавит или отнимет значение a из k-й ячейки массива коэффициентов:

```
P6: if Op = '+' then
       P.a[k] := P.a[k] + a
    else
       P.a[k] := P.a[k] - a;
```

Было бы неправильно просто записывать значение коэффициента в `a[k]`, поскольку в записи многочлена могут быть несколько членов с одинаковой степенью x — синтаксис этого не запрещает. Складывая или вычитая каждый коэффициент с предыдущей суммой, транслятор как бы приводит подобные.

Определение степени n многочлена P выполняет семантическая процедура P_7.

```
P7: P.n := nmax;
    while (P.n > 0) and (P.a[P.n] = 0) do
       P.n := P.n-1;
```

Теперь, пользуясь диаграммой и вставляя в текст анализатора в соответствующих местах определенные нами семантические процедуры, можно записать распознаватель многочлена.

```
{ Многочлен }
procedure Poly(var P: tPoly);
var
   a     : integer;  { Модуль коэффициента }
   k     : integer;  { Степень слагаемого  }
   Op    : char;     { Знак операции       }
begin
   ClearPoly(P);                              { P3 }
   if Ch in ['+','-'] then begin
      Op := Ch;                               { P4 }
      NextCh;
      end
   else
      Op := '+';                              { P5 }
   Addend(a, k);
   if Op = '+' then                           { P6 }
      P.a[k] := P.a[k] + a                    {    }
   else                                       {    }
      P.a[k] := P.a[k] - a;                   {    }
   while Ch in ['+','-'] do begin
      Op := Ch;                               { P4 }
      NextCh;
      Addend(a, k);
      if Op = '+' then                        { P6 }
         P.a[k] := P.a[k] + a                 {    }
      else                                    {    }
         P.a[k] := P.a[k] - a;                {    }
   end;
   P.n := nmax;                               { P7 }
   while (P.n > 0) and (P.a[P.n] = 0) do      {    }
      P.n := P.n-1;                           {    }
end;
```

Задача транслятора слагаемого, как уже говорилось, состоит в определении коэффициента (точнее, модуля коэффициента) a и степени k слагаемого. В соответствии с этим предусмотрим и необходимые семантические процедуры (рис. 2.32).

Процедура P_8 присваивает коэффициенту a значение, равное величине целого числа, которое вычисляется транслятором целого.

```
P8:   a := значение целого;
```

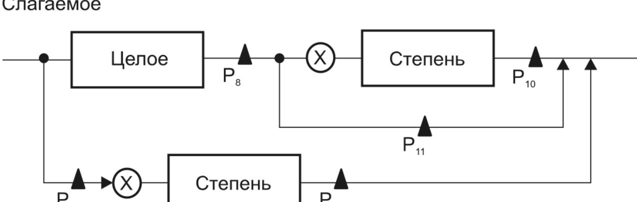


Рис. 2.32. Синтаксическая диаграмма слагаемого с семантическими процедурами

В программе это будет реализовано подстановкой переменной *a* в качестве параметра при вызове процедуры `Number`.

Если коэффициент отсутствует, он принимается равным единице.

```
P9:    a := 1;
```

Значение степени либо берется равным вычисленному распознавателем степени, либо нулевым, если в записи слагаемого отсутствует *x*.

```
P10:    k := значение степени;
P11:    k := 0;
```

Теперь записать распознаватель-транслятор слагаемого не составляет труда.

```
{ Слагаемое }
procedure Addend(var a, k : integer);
begin
   if Ch in ['0'..'9'] then begin
      Number(a);                    { P8 }
      if Ch = 'x' then begin
         NextCh;
         Power(k);                  { P10 }
         end
      else
         k := 0;                    { P11 }
      end
   else if Ch = 'x' then begin
      a := 1;                       { P9 }
      NextCh;
      Power(k);                     { P10 }
      end
   else
      Error('Ожидается число или "x"');
end;
```

Реализация распознавателя нетерминала "Степень" не вызывает затруднений и выполняется по соответствующей синтаксической диаграмме, на которой обозначены семантические процедуры P_{11} и P_{12} (рис. 2.33). Чтобы защитить программу от ошибки при обращении к массиву коэффициентов, в процедуре P_{11} предусмотрен контроль величины показателя степени. Вычисляемая степень обозначена p.

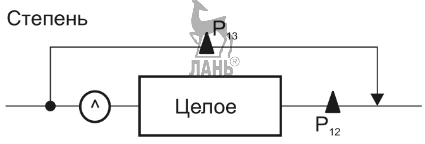


Рис. 2.33. Синтаксическая диаграмма степени с семантическими процедурами

```
P12:    p := значение целого;
        if p > nmax then
           Error('Слишком большая степень');
P13:    p := 1;
```

Обратите внимание, что в случае, когда семантическая процедура (P_{13}) расположена на ветви диаграммы, где не было терминальных и нетерминальных блоков, она сама играет роль блока, что несколько меняет структуру программы-распознавателя (в нашем случае появляется ветвь **else**).

```
{ Степень }
procedure Power(var p : integer);
begin
   if Ch = '^' then begin
      NextCh;
      Number(p);                               { P12 }
      if p > nmax then                         {     }
         Error('Слишком большая степень');     {     }
      end
   else
      p := 1;                                  { P13 }
end;
```

Обработка ошибок при трансляции

В рассмотренных примерах обработка ошибок выполнялась с помощью процедуры `Error`, которая после выдачи сообщения прекращала работу всей программы. Такой способ реакции на ошибку

упрощает синтаксический анализатор, но далеко не идеален. Можно рассмотреть несколько вариантов его усовершенствования:

- Завершение работы распознавателя (транслятора) после обнаружения первой ошибки без прерывания работы вызвавшей его программы.
- Продолжение работы распознавателя после обнаружения первой ошибки с целью обнаружения других.

Второй вариант реакции на ошибки используется во многих трансляторах языков программирования. В литературе можно найти немало рекомендаций по способам восстановления распознавателя после обнаружения ошибки с целью продолжения анализа [Грис, 1975], [Хантер, 1984], [Вирт, 1985], [Wirth, 1996], [Ахо, 2008]. Однако, общего решения задачи не существует. Многое зависит от конкретного языка. Качественное восстановление, обеспечивающее выдачу осмысленных сообщений, предполагает, в том числе, использование эмпирических приемов. Рассмотрение таких подходов выходит за рамки этой книги. Любознательным читателям могу рекомендовать посмотреть названные источники или проделать собственные опыты.

Обсудим и реализуем более простой первый вариант. Он совсем неплох. В условиях, когда скорость работы компьютеров неизмеримо возросла, время, необходимое для компиляции программы[55], невелико, и его потери, связанные с тем, что компилятор не сообщил программисту о нескольких ошибках сразу, исчезающе малы.

Итак, необходимо организовать реакцию на ошибку таким образом, чтобы после выдачи сообщения, анализатор не останавливал всю программу, а лишь завершил свою работу, давая возможность продолжить исполнение вызвавшей его программе.

Первый вариант усовершенствования состоит в том, чтобы после вызова каждой распознающей процедуры проверять успешность ее завершения и, если при анализе нетерминала обнаружена ошибка, пропускать оставшиеся части вызывающей процедуры. В этом случае процедура `Error` уже не прерывает программу с помощью `Halt`, а формирует признак ошибки, который может быть проверен в распознающих процедурах. Такой подход, однако, сильно загромоздит программы анализатора, сделает их неудобочитаемыми.

[55] Точнее, одной единицы компиляции, например, модуля.

Используем другое решение. Проблема при обработке ошибок состоит лишь в том, чтобы после обнаружения ошибки завершить *все процедуры*, цепочка вызова которых привела к процедуре, обнаружившей ошибку. Это можно сделать, если при обнаружении ошибки процедура `Error` установит признак ошибки и прочитает текст до конца. Текущим символом станет "конец текста". Поскольку этот символ не может встретиться в анализируемом тексте, он будет отвергнут всеми частями анализатора, который завершит свою работу без прерывания программы в целом. В качестве признака ошибки разумно использовать номер ошибочного символа (обозначим `ErrPos`), ненулевое значение которого соответствует наличию ошибки и несет информацию о ее местоположении. Перед началом работы анализатора до первого вызова `NextCh` нужно выполнить `ErrPos := 0`. Исправленные процедуры `Error` и `NextCh` теперь выглядят так:

```
{ Ошибка }
procedure Error(Message : string);
   { Message - сообщение }
var
   e : integer;
begin
   if ErrPos = 0 then begin
      WriteLn('^': Pos);
      WriteLn('Синтаксическая ошибка: ', Message);
      e := Pos;
      while Ch <> EOT do NextCh;
      ErrPos := e;
   end;
end;
```

Процедура `Error` «самоблокируется», проверяя значение `ErrPos` и обходя выдачу сообщений при повторных вызовах. Пользоваться ею можно так же, как и раньше, как бы считая, что после вызова `Error` работа анализатора прерывается. Дополнительные проверки на наличие ошибки после обращения к распознавателям нетерминалов не нужны.

`NextCh` после обнаружения ошибки всегда возвращает «конец текста».

```
{ Читать следующий символ }
procedure NextCh;
```

```
begin
   if ErrPos <> 0 then
      Ch := EOT
   else
      …
end;
```

Надо, однако, беспокоиться о том, чтобы при обнаружении ошибки не оставались неопределенными или недопустимыми данные, связанные с семантической обработкой. Они могут использоваться в вычислениях, которые будут выполняться после выдачи сообщения об ошибке при возврате из распознающих процедур. При этом не должно возникнуть недопустимых ситуаций: деление на ноль, разыменование неопределенного или равного **nil** указателя, выход индекса за границы массива и т. п. Например, распознаватель степени, в задачу которого входит определение величины *p*, при использовании нашей технологии следует записать так:

```
{ Степень }
procedure Power(var p : integer);
begin
   if Ch = '^' then begin
      NextCh;
      Number(p);
      if p > nmax then begin
         Error('Слишком большая степень');
         p := 0;{Степень не должна остаться
                 слишком большой}
      end
      end
   else
      p := 1;
end;
```

Требуется еще одно уточнение. Поскольку по завершении работы распознавателя, обнаружившего ошибку, текущим символом станет EOT, его проверка уже не будет достаточным условием успешного окончания анализа. Использовавшееся выше завершение:

```
if Ch <> EOT then
   Error('Ожидается конец текста')
else
   WriteLn('Правильно');
```

должно быть заменено следующим:

```
 if (ErrPos = 0) and (Ch = EOT) then
    WriteLn('Правильно')
 else
    Error('Ожидается конец текста');
```

Существует мнение, что лучшим вариантом выхода из состояния ошибки в анализаторе, работающем по алгоритму рекурсивного спуска, является использование исключений, обработка которых предусмотрена в некоторых языках программирования. Вы можете поэкспериментировать и составить свое суждение на этот счет.

Табличный *LL*(1)-анализатор

Рассматривая автоматные языки, мы использовали детерминированный конечный автомат в роли эффективного универсального распознавателя. Один из вариантов его реализации — программная интерпретация таблицы переходов автомата. На похожих принципах может быть построен и распознаватель для *LL*(1)-грамматик, который будет представлять собой реализацию определенного подкласса упоминавшихся выше МП-автоматов.

Вначале модифицируем таблицу переходов конечного автомата. Ее обычный формат таков:

Таблица 2.4. Таблица переходов конечного автомата

Состояние	**Символ**				
	a	**b**	**c**	**d**	…
1					
2					
3					
4					
…					

Все состояния пронумерованы. В колонках символов для примера записаны буквы из начала латинского алфавита. Для реальных языков количество допустимых символов может быть довольно большим. Если же рассматривать практические аспекты реализации, то придется учесть, что на вход программы, моделирующей автомат, может поступать вообще любой символ.

Чтобы сократить размер таблицы, будем размещать различные символы в одном столбце. В каждом состоянии будет проверяться

совпадение с единственным символом. Число состояний при этом может увеличиться.

Для примера возьмем конечный автомат, распознающий целые числа без знака. Таблица 2.5, имеющая традиционный вид, задает его переходы.

Таблица 2.5. Таблица переходов КА, распознающего целые без знака

Состояние	**Символ**	
	Цифра	**Не цифра**
1	2	*E*
2	2	*K*
E	*E*	*E*
K		

Здесь *E* означает состояние ошибки, а *K* — конечное состояние автомата.

Следующая таблица 2.6 имеет модифицированный вид. Символы записываются во втором столбце, состояние в которое переходит автомат при совпадении входного символа и символа в таблице — в третьем. В четвертом столбце отмечено, возникает ли ошибка, если входной символ не совпадает с символом, записанным в таблице для данного состояния. Если в графе «Ошибка» записано «Нет», то при несовпадении символов автомат переходит в следующее по порядку состояние.

Таблица 2.6. Модифицированная таблица переходов КА, распознающего целые

Состояние	**Символ**	**Переход**	**Ошибка**
10	Цифра	11	Да
11	Цифра	11	Нет
12	Любой	0	Нет

По причинам, которые мы вскоре выясним, состояния нового автомата пронумерованы не с 1. Переход в состояние 0, который происходит при получении автоматом, находящемся в состоянии 12, любого символа (например, символа «конец текста») означает завершение его работы с принятием (части) входной цепочки.

Автомат, распознающий целые, можно попробовать применить при построении автомата, распознающего «Степень» из примера про многочлены (см. рис. 2.33), а также многочлен в целом. Начнем с автома-

та, распознающего степень. Построим таблицу переходов (табл. 2.7). Пусть состояния этого автомата начинаются с 20.

Таблица 2.7. Таблица переходов распознавателя степени

Состояние	**Символ**	**Переход**	**Ошибка**	**Вызов**	**Читать**
20	^	22	Нет	Нет	Да
21	Любой	0	Нет	Нет	Нет
22	Любой	10	Нет	Да	Нет
23	Любой	0	Нет	Нет	Нет

Потребовалось также добавить два новых столбца. Столбец «Читать» управляет чтением следующего символа: если в этом столбце стоит «Да», то при совпадении входного символа и символа, записанного во втором столбце для данного состояния, читается следующий символ входной цепочки. В предыдущей таблице 2.5 предполагалось, что при совпадении следующий символ считывается всегда.

Распознавание начинается, когда автомат находится в состоянии 20. Если текущим символом является «^», автомат переходит в состояние 22, считывая следующий символ. Если первый символ не равен «^» автомат переходит в состояние 21 и принимает часть входной цепочки, за которую он отвечает — запись степени в этом случае пуста.

Попав в состояние 22, автомат ожидает целое число. Распознаватель целого в нашем примере начинается с состояния 10 (нумерация не с единицы, чтоб подчеркнуть, что этот распознаватель не составляет законченный автомат, а только его часть). Поэтому в состоянии 22 запрограммирован переход в состояние 10. Но это не обычный переход, а переход с возвратом. По окончании распознавания целого должен произойти возврат из состояния 12 в состояние 23. Чтобы отметить особенность таких переходов, в таблицу введена графа «Вызов», в которой записывается «Да», если происходит переход с возвратом. Автомат, распознающий степень, как бы вызывает автомат, распознающий целое. Нетрудно догадаться, что суть происходящего подобна вызову одной подпрограммы из другой.

Значение 0 в графе «Переход» следует теперь воспринимать как возврат в состояние, следующее за тем, из которого произошел вызов. Важно понимать, что при разных вызовах это может быть разное состояние. Например, вызовы целого могут выполняться распознавателем степени, а могут — распознавателем слагаемого.

Табличный транслятор многочленов

Пользуясь выработанным форматом таблицы переходов, заполним ее для распознавателя многочленов (который будет, как свои части, включать уже рассмотренные распознаватели целого и степени). Имея в виду, что должен выполняться не только синтаксический анализ, но и трансляция многочлена, предусмотрим в таблице графу «Процедура», в которой будем записывать номер семантической процедуры (из числа предусмотренных раньше для примера про многочлены). Эта процедура будет вызываться при совпадении входного символа и символа в таблице. Если указан нулевой номер семантической процедуры, вызов не происходит (или процедура P_0 не выполняет никаких действий). Некоторые состояния добавляются в таблицу лишь для того, чтобы предусмотреть в нужные моменты выполнение необходимых семантических процедур (например, состояния 1, 6, 25).

Таблица 2.8. Таблица переходов *LL*(1)-распознавателя многочленов

Сост.	**Симв.**	**Перех.**	**Ош.**	**Выз.**	**Чит.**	**Проц.**	**Примечание**
Многочлен							
1	Л	2	–	–	–	3	Обнуление коэффициентов
2	+	5	–	–	+	4	
3	–	5	–	–	+	4	
4	Л	5	–	–	–	5	Нет знака спереди
5	Л	12	–	+	–	0	На 1-е слагаемое
6	Л	7	–	–	–	6	После 1-го слагаемого
7	+	10	–	–	+	4	Начало цикла
8	–	10	–	–	+	4	
9	⊥	0	+	–	–	7	Выход
10	Л	12	–	+	–	0	На слагаемое
11	Л	7	–	–	–	6	Конец цикла
Слагаемое							
12	Ц	15	–	–	–	0	
13	x	21	+	+	+	10	$a = 1$
14	Л	0	–	–	–	9	После степени
15	Л	25	–	+	–	0	На целое
16	Л	17	–	–	–	8	После целого

Продолжение табл. 2.8

Сост.	Симв.	Перех.	Ош.	Выз.	Чит.	Проц.	Примечание
17	x	19	–	–	+	0	
18	Л	0	–	–	–	11	$k = 0$
19	Л	21	–	+	–	0	На степень
20	Л	0	–	–	–	9	Конец слагаемого
Степень							
21	^	23	–	–	+	0	
22	Л	0	–	–	–	13	$p = 1$
23	Л	25	–	+	–	0	На целое
24	Л	0	–	–	–	12	
Целое							
25	Л	26	–	–	–	1	Инициализация
26	Ц	27	+	–	+	2	Первая цифра
27	Ц	27	–	–	+	2	Последующие цифры
28	Л	0	–	–	–	0	

Чтобы таблица была компактней, вместо «Да» и «Нет» в ней используются «+» и «–», «Л» обозначает любой символ, «Ц» — любую цифру, «⊥» — символ «конец текста».

Перед выполнением перехода с возвратом необходимо запоминать номер состояния, в которое автомат должен возвратиться. Недостаточно использовать для этого отдельную переменную, способную в каждый момент хранить номер только одного состояния. Дело в том, что вызовы могут быть вложенными: «Многочлен» вызывает «Слагаемое», «Слагаемое» вызывает «Целое» и «Степень», «Степень» — «Целое». Вызов, произошедший последним, закачивается первым. Это соответствует стековой дисциплине «последним пришел, первым ушел» (LIFO — Last In, First Out). Для запоминания номера состояния перед выполнением перехода с возвратом нужен стек.

LL(1)-драйвер

Программу, которая выполняет распознавание, интерпретируя таблицу, называют драйвером. В нашей реализации драйвера будет использована таблица, тип которой определим как массив из записей.

```
tSynTable = array [1..n] of record
   Ch     : char;       { Символ     }
```

```
   Go    : integer;  { Переход   }
   Err   : Boolean;  { Ошибка    }
   Call  : Boolean;  { Вызов     }
   Read  : Boolean;  { Читать    }
   Proc  : integer;  { Процедура }
end;
```

Драйвер использует следующие переменные:

```
T     : tSynTable;   { Таблица          }
Ch    : char;        { Текущий символ   }
Err   : integer;     { Признак ошибки   }
Stack : tStack;      { Стек             }
i     : integer;     { Номер состояния }
```

Приведенный в листинге 2.4 текст программы-драйвера не включает часть, которая заполняет таблицу перед началом работы. Программа содержит и другие условности, касающиеся обозначений произвольного символа и цифр в таблице.

Листинг 2.4. Драйвер табличного *LL*(1) анализатора

```
ResetText;
Init(Stack);
Push(Stack, 0);
Err := 0;
i := 1;
repeat
   if ( Ch=T[i].Ch ) or
      ( T[i].Ch = Любой ) or
      ( T[i].Ch = Цифра ) and (Ch in ['0'..'9'])
   then begin
      if T[i].Proc <> 0 then Proc(T[i].Proc);
      if T[i].Read then NextCh;
      if T[i].Go = 0 then
         Pop(Stack, i)
      else begin
         if T[i].Call then Push(Stack, i+1);
         i := T[i].Go;
      end;
      end
   else if T[i].Err then
      Err := i
   else
      i := i+1;
until (i=0) or (Err<>0);
```

Предполагается, что процедура `Proc` способна выполнять семантическую процедуру с заданным номером.

Драйвер завершает работу в двух случаях: когда в результате выполнения очередного возврата извлечен ноль из стека (он помещается в стек перед циклом), и при возникновении ошибки. Переменная `Err` получает значение, равное номеру состояния, в котором возникла ошибка.

Использование множеств символов

В ряде случаев одинаковые действия распознавателя выполняются для набора символов. Например, в задаче про многочлены знаки «+» и «–» всегда обрабатываются одинаково (строки 2, 3 и 7, 8 в табл. 2.8). Используя в таблице вместо отдельных символов множества, можно уменьшить число состояний (табл. 2.9).

```
tSynTable = array [1..n] of record
   ChSet : set of char; { Символы   }
   Go    : integer;     { Переход   }
   Err   : Boolean;     { Ошибка    }
   Call  : Boolean;     { Вызов     }
   Read  : Boolean;     { Читать    }
   Proc  : integer;     { Процедура }
end;
```

Таблица 2.9. Таблица переходов с множествами символов

Сост.	Символы	Пер.	Ош.	Выз.	Чит.	Проц.	Примечание
Многочлен							
1	[#0..#255]	2	–	–	–	3	Обнуление коэффициентов
2	['+', '–']	4	–	–	+	4	
3	[#0..#255]	4	–	–	–	5	Нет знака спереди
4	[#0..#255]	10	–	+	–	0	На 1-е слагаемое
5	[#0..#255]	6	–	–	–	6	После 1-го слагаемого
6	['+', '–']	8	–	–	+	4	Начало цикла
7	[⊥]	0	+	–	–	7	Выход
8	[#0..#255]	10	–	+	–	0	На слагаемое
9	[#0..#255]	6	–	–	–	6	Конец цикла
Слагаемое							
10	['0'..'9']	13	–	–	–	0	
11	['x', 'X']	19	+	+	+	10	a = 1

Продолжение табл. 2.9

Сост.	Символы	Пер.	Ош.	Выз.	Чит.	Проц.	Примечание
12	[#0..#255]	0	–	–	–	9	После степени
13	[#0..#255]	23	–	+	–	0	На целое
14	[#0..#255]	15	–	–	–	8	После целого
15	['x', 'X']	17	–	–	+	0	
16	[#0..#255]	0	–	–	–	11	k = 0
17	[#0..#255]	19	–	+	–	0	На степень
18	[#0..#255]	0	–	–	–	9	После степени
Степень							
19	['^']	21	–	–	+	0	
20	[#0..#255]	0	–	–	–	13	p = 1
21	[#0..#255]	23	–	+	–	0	
22	[#0..#255]	0	–	–	–	12	
Целое							
23	[#0..#255]	24	–	–	–	1	Инициализация
24	['0'..'9']	25	+	–	+	2	Первая цифра
25	['0'..'9']	25	–	–	+	2	Последующие цифры
26	[#0..#255]	0	–	–	–	0	

Множество [#0..#255] включает все символы, начиная с символа, имеющего порядковый номер 0, и заканчивая символом с порядковым номером 255.

Число состояний в нашем примере удалось уменьшить всего на два. Но получен еще ряд преимуществ: оказалось очень легко разрешить использование как строчной, так и прописной буквы *x* в записи многочлена (см. строку 15 в табл. 2.8), устранены условности, связанные с обозначением произвольного символа и множества цифр.

Использование множеств увеличивает расход памяти на представление таблицы. В случае если во второй графе таблицы помещается символ, он занимает один байт (при однобайтовой кодировке символов), если множество из символов — необходимо 32 байта для хранения каждого такого множества (1 бит на символ * 256 различных символов / 8 бит в одном байте).

Листинг 2.5. Драйвер табличного *LL*(1)-анализатора, использующий множества

```
ResetText;
Init(Stack);
Push(Stack, 0);
Err := 0;
i := 1;
repeat
   if Ch in T[i].ChSet then begin
      if T[i].Proc <> 0 then Proc(T[i].Proc);
      if T[i].Read then NextCh;
      if T[i].Go = 0 then
         Pop(Stack, i)
      else begin
         if T[i].Call then Push(Stack, i+1);
         i := T[i].Go;
      end;
      end
   else if T[i].Err then
      Err := i
   else
      i := i+1;
until (i=0) or (Err<>0);
```

Можно видеть, что программа-драйвер (листинг 2.5) совершенно не зависит от анализируемого языка. Это универсальный *LL*(1)-распознаватель (и транслятор). Синтаксис конкретного языка целиком определяется таблицей.

Обработка ошибок

Реакция на синтаксическую ошибку при использовании табличного анализатора может быть организована просто и естественно. Действительно, при обнаружении ошибки цикл анализа немедленно прекращается, а переменная `Err` при этом содержит номер состояния, в котором произошла ошибка. Значение `Err` однозначно определяет тип ошибки. Сообщения об ошибке могут быть помещены в отдельную графу таблицы, в те её строки, которые содержат «+» в столбце «Ошибка». Можно генерировать сообщения автоматически. Каждое такое сообщение может состоять из слова «Ожидается» с последующим перечислением множества символов `T[Err].ChSet`.

Кроме синтаксических необходимо обрабатывать ошибки, которые обнаруживаются семантическими процедурами. При использовании табличного распознавателя организовать реакцию на них также не-

сложно. Одно из решений, позволяющее не менять программу-драйвер — присваивать переменной `Err` отрицательное значение при обнаружении ошибки семантической процедурой.

Рекурсивный спуск и табличный анализатор

Несмотря на совершенно разное устройство программ, принцип работы алгоритма рекурсивного спуска и табличного распознавателя похожи. Таблица состоит из частей, соответствующих нетерминалам грамматики, подобно тому, как программа, использующая рекурсивный спуск, состоит из распознающих процедур. Драйвер, работающий по таблице, реализует те же механизмы, которые действуют при вызовах процедур, в том числе рекурсивных. Используемый драйвером явно стек, неявно присутствует при вызовах процедур.

Выбирая один из способов построения распознавателя, можно принимать во внимание следующие соображения.

- При табличном анализе лучше решается проблема выхода из состояния ошибки.
- Табличный анализатор допускает использование языка, не содержащего рекурсивных процедур.

Если речь идет о ручном программировании рекурсивного спуска и составлении таблицы разбора также вручную, то оказывается, что подготовка таблицы — это тоже своеобразное программирование на особом языке, предусматривающем условные переходы и переходы с возвратом[56].

- Программа-анализатор, написанная по методу рекурсивного спуска, наглядней таблицы. Программирование вручную рекурсивного спуска проще ручного составления таблицы.

Независимость программы-драйвера от входного языка должна, по идее, давать преимущества и создавать предпосылки для создания более гибких систем. В действительности, независимой от языка оказывается только часть, отвечающая за синтаксис. В то время как семантические процедуры, которые встраиваются в программу-транслятор, будут разными для различных языков.

- Независимость $LL(1)$-драйвера от входного языка создает преимущество при решении задачи синтаксического анализа. При наличии семантической обработки независимость теряется.

---

[56] Очень похоже на программирование на примитивных диалектах Бейсика с помощью IF, GOTO, GOSUB и нумерованных строк.

- Добавление семантической обработки к анализатору, написанному методом рекурсивного спуска, проще и естественней в силу лучшей структуры и наглядности такого анализатора.

  Простое устройство синтаксической таблицы создает возможность ее автоматического формирования по описанию грамматики с помощью специальных программ — генераторов анализаторов. Однако с не меньшим успехом генератор анализаторов может порождать не таблицу, а программу на языке программирования, работающую по методу рекурсивного спуска. Проблемы, которые он при этом должен будет решать, будут в основном те же.
- Рекурсивный спуск и табличный $LL(1)$-анализ предоставляют сопоставимые возможности для автоматического построения анализатора.
- Анализатор, работающий по методу рекурсивного спуска, будет, скорее всего, быстрее табличного, интерпретирующего переходы по таблице, в то время как аналогичные переходы при рекурсивном спуске выполняются скомпилированной программой.

Трансляция выражений

Выражение — одно из основных понятий языков программирования. Подсистема, ответственная за трансляцию выражений, представляет собой важную и одну из самых сложных частей компилятора.

Польская запись

Обычная форма выражений предусматривает запись знака операции между операндами. Например:

$$a + b,$$
$$a + b*c,$$
$$(a + b)*c.$$

Такую запись называют *инфиксной*.

В некоторых случаях используется префиксная (прямая польская) запись, когда обозначения операций записываются до операндов:

$$+\ a\ b,$$
$$+\ a\ *\ b\ c,$$
$$*\ +\ a\ b\ c.$$

Постфиксная (обратная польская) запись требует, чтобы знаки операций записывались после соответствующих операндов. Приведенные выше инфиксные выражения в обратной польской записи будут выглядеть так:

$$a\ b\ +,$$
$$a\ b\ c\ *\ +,$$
$$a\ b\ +\ c\ *.$$

Польской такая нотация названа в честь ее изобретателя — польского математика и логика Яна Лукасевича (Jan Łukasiewicz, 1878–1956). В этой книге обратная польская запись будет называться также *польской инверсной записью* (сокращенно ПОЛИЗ).

Польская запись не содержит скобок.

При записи в ПОЛИЗ операнды записываются в порядке следования в исходном выражении, операции — после своих операндов в порядке выполнения.

Таблица 2.10. Примеры записи выражений в обратной польской записи

Обычная (инфиксная) запись	Обратная польская запись
$(a + b)/(c + d)$	$a\ b + c\ d + /$
$a + b*c - a/(a + b)$	$a\ b\ c * + a\ a\ b + / -$
$\sqrt{1-\sin^2 x}$	1 x sin ² – √

В последнем примере в таблице 2.10 «sin», «²» и «√» следует рассматривать как знаки одноместных операций вычисления синуса, квадрата и квадратного корня соответственно.

Интерес к ПОЛИЗ обусловлен тем, что она удобна для вычисления выражений и как промежуточная форма представления выражений в трансляторе.

Принцип обратной польской записи может быть применен не только к выражениям, но и операторам языков программирования.

Алгоритм вычисления выражений в обратной польской записи

Выражения в обратной польской записи вычисляются с помощью стека. При чтении ПОЛИЗ слева направо операнды помещаются в стек, а операции применяются к верхним элементам стека. Результат выполнения операции возвращается в стек, заменяя собою операнды. По исчерпании выражения его значение находится на вершине стека.

В приведенном алгоритме (листинг 2.6) предполагается, что ПОЛИЗ состоит из отдельных *элементов*, и мы в состоянии отличать операнды от операций (с помощью множеств «Операнды», «Одноместные операции», «Двуместные операции»), а одноместные операции от двуместных. Предполагается также, что имеются только одноместные и двуместные операции[57].

Листинг 2.6. Алгоритм вычисления выражения в ПОЛИЗ

```
Читать(элемент); { Первый элемент ПОЛИЗ }
while элемент <> конец do begin
   if элемент in Операнды then
      Push(Значение(элемент))
   else if элемент in Одноместные операции then begin
      Pop(x);
      Push( Оп1(элемент, x) );
      end
   else if элемент in Двуместные операции then begin
      Pop(x2); Pop(x1);            {Обратите внимание }
      Push(Оп2(элемент, x1, x2));{на порядок x1 и x2}
   end;
   Читать(элемент);
end;
Pop(Значение выражения);
```

Операции со стеком в этом алгоритме выполняются с помощью процедур `Push` (в стек) и `Pop` (из стека), у которых параметр-стек для краткости не указан.

Функция `Значение` позволяет по записи элемента-операнда получить его значение. В зависимости от того, как кодируются операнды, является ли операнд константой или переменной, эта функция реализуется по-разному и здесь не детализируется.

Значением функций `Оп1` и `Оп2` является результат применения одноместной и двуместной операции, заданной значением первого параметра, к последующим параметрам. Например, если речь идет об операциях с вещественными, эти функции могут быть такими:

```
function Оп2(элемент: Элементы; x1, x2: real): real;
begin
   case элемент of
```

[57] В общем случае, конечно, встречаются и многоместные операции. В этой роли могут выступать, например, функции нескольких переменных. Распространить алгоритм на этот случай не составляет труда.

```
   сложение:  Оп2 := x1 + x2;
   вычитание: Оп2 := x1 - x2;
   умножение: Оп2 := x1*x2;
   деление:   Оп2 := x1/x2;
   end;
end;

function Оп1(элемент: Элементы; x: real): real;
begin
   case элемент of
   изменение знака: Оп1 := -x;
   синус:           Оп1 := sin(x);
   косинус:         Оп1 := cos(x);
   end;
end;
```

Схема трансляции выражений

Разные варианты использования обратной польской записи при трансляции выражений показаны на рисунке 2.34.

Быстрый интерпретатор (рис. 2.34*а*) вначале преобразует выражение в ПОЛИЗ, а затем вычисляет его по алгоритму, приведенному выше.

Компилятор (рис. 2.34*б*) использует ПОЛИЗ в роли промежуточного представления, по которому генератор формирует машинный код.

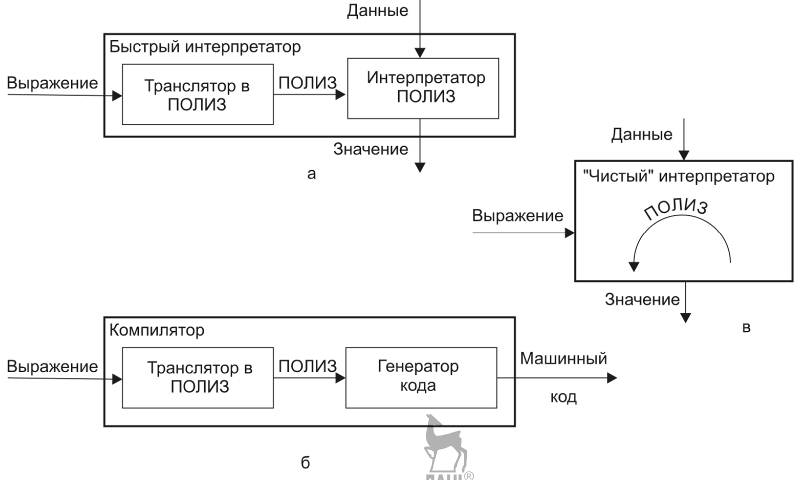


Рис. 2.34. Схемы трансляции выражений

В схеме «чистого интерпретатора» польская запись в явном виде не формируется, но совершаемые интерпретатором действия включают в себя те, что нужны для получения польской записи и те, что служат для вычисления с помощью стека.

Перевод выражений в обратную польскую запись

Преобразование инфиксного выражения в польскую запись может выполняться в ходе его синтаксического анализа. Чтобы определить действия, которые для этого нужны, достаточно обозначить на синтаксических диаграммах выражения семантические процедуры (рис. 2.35).

Определяя семантические процедуры, будем, как и всегда, считать, что переменная `Ch` содержит текущий символ входной цепочки. Вспомогательная переменная `Op` имеет символьный тип и способна хранить знак операции.

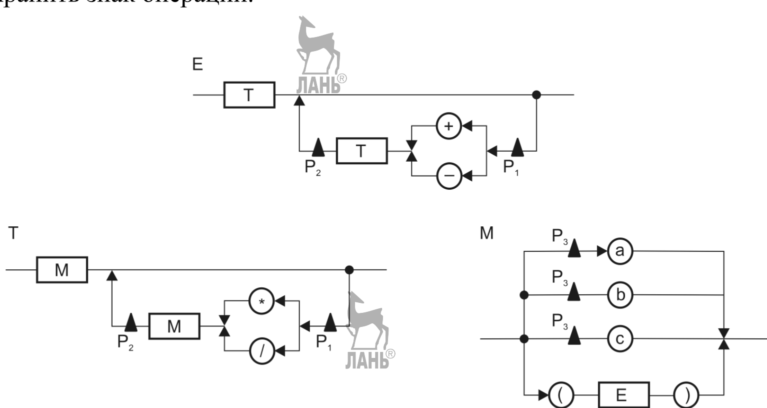


Рис. 2.35. Семантические процедуры для перевода выражения в обратную польскую запись

Задача же состоит просто в том, чтобы вывести обратную польскую запись выражения.

P_1: `Запомнить(Ch);`
P_2: `Вспомнить(Op); Write(Op);`
P_3: `Write(Ch);`

Реализация действий, обозначенных словами «Запомнить» и «Вспомнить», может быть различна, но в любом случае для запоми-

нания знака операции необходимо использовать стек. Это обусловлено тем, что выражения (слагаемые, множители) могут быть вложенными и в определенные моменты работы алгоритма запомненными могут быть сразу несколько знаков. При этом запомненный первым должен быть «вспомнен» последним.

Если используется табличный распознаватель, то стек присутствует явно, и действие `Запомнить(Ch)` заменяется на `Push(Ch)`, а `Вспомнить(Op)` — на `Pop(Op)`.

Если применяется рекурсивный спуск, то достаточно сохранять знак операции в локальных переменных распознающих процедур «Выражение» и «Слагаемое». Не записывая программу целиком, приведем лишь процедуру для распознавания слагаемого.

```
procedure T;
var
   Op: char;
begin
   M; { Множитель }
   while Ch in ['+', '-'] do begin
      Op := Ch;
      NextCh;
      M;
      Write(Op);
   end;
end;
```

Алгоритм Э. Дейкстры перевода выражений в обратную польскую запись (метод стека с приоритетами)

Не следует думать, что синтаксически управляемая трансляция выражений в ПОЛИЗ, рассмотренная выше, — единственный способ такого перевода. Существуют и другие методы. Рассмотрим простой и эффективный алгоритм Э. Дейкстры, основанный на использовании стека операций. Он состоит в следующем:

1. Каждому знаку операции присваивается числовой приоритет. Операции, выполняемые в первую очередь, имеют больший приоритет. Приоритеты операций начинаются с 2.
2. Открывающим скобкам присваивается приоритет 0, закрывающим — 1.
3. Входная строка считывается слева направо. Операнды по мере их считывания помещаются в выходную строку.

4. Знаки перед записью в выходную строку помещаются в стек (рис. 2.36) в соответствии со следующей дисциплиной:

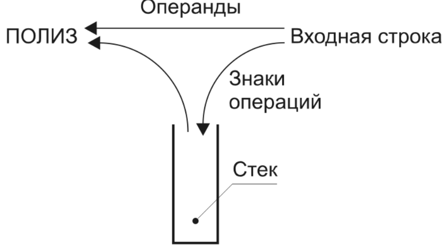


Рис. 2.36. Использование стека в алгоритме Э. Дейкстры

- Открывающая скобка (знак с приоритетом 0) помещается в стек.
- Если приоритет знака операции больше приоритета знака на вершине стека или стек пуст, новый знак добавляется в стек.
- Если приоритет знака меньше или равен приоритету знака на вершине стека, из стека в выходную строку «выталкиваются» все знаки с приоритетами большими или равными приоритету входного знака. После этого входной знак записывается в стек.
- Закрывающая скобка выталкивает в выходную строку все знаки до открывающей скобки. Открывающая скобка удаляется из стека. Закрывающая и открывающая скобки в выходную строку не записываются.

5. После просмотра всех символов входной строки из стека выталкиваются оставшиеся знаки.

Этот метод может быть распространен и на функции, переменные с индексами и др.

Интерпретация выражений

Рассмотрим, каким образом вычисляется выражение по его записи, без явного преобразования в ПОЛИЗ. Очевидно, что можно действовать, комбинируя действия, связанные с переводом в ПОЛИЗ с действиями, которые выполняются при вычислении выражения по его польской записи. Вместо вывода операндов и операций в выходную строку надо помещать операнды в стек, а операции выполнять, применяя их к верхним элементам стека и возвращая на стек результат.

Способ решения задачи определен с помощью синтаксических диаграмм (рис. 2.37) и семантических процедур. В отличие от предыду-

щих примеров, добавлена возможность записывать знак перед первым слагаемым.

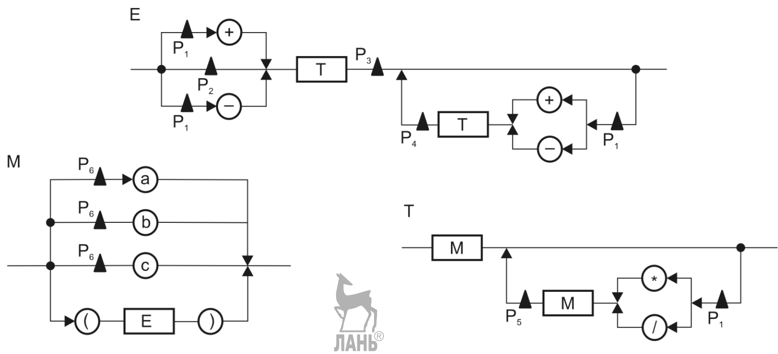


Рис. 2.37. Семантические процедуры для интерпретации выражений

```
P1: Запомнить(Ch);
P2: Запомнить('+');
P3: Вспомнить(Op);
      if Op = '-' then begin
         Pop(x); Push(-x);
      end;
P4: Вспомнить(Op);
      Pop(x2); Pop(x1);
      if Op = '+' then  Push(x1 + x2)
      else {Op = '-'}   Push(x1 – x2);
P5: Вспомнить(Op);
      Pop(x2); Pop(x1);
      if Op = '*' then  Push(x1*x2)
      else {Op = '/'}   Push(x1/x2);
P6: Push(Значение(Ch));
```

Здесь процедуры Push и Pop оперируют стеком операндов, в то время как Запомнить и Вспомнить явно или неявно обращаются к стеку операций.

Семантическое дерево выражения

Как уже говорилось, задача разбора состоит в построении дерева вывода (синтаксического дерева). Однако, ни один из рассмотренных нами алгоритмов не строит это дерево в явном виде. Структура входной цепочки, которую представляет дерево, лишь отражается в по-

следовательности действий, совершаемых синтаксическим анализатором.

На практике важную роль может играть семантическое дерево. Его построение облегчает решение многих задач трансляции. Дерево может использоваться в качестве промежуточного представления программы.

Рассмотрим построение и использование семантического дерева на примере простых арифметических выражений, которые уже неоднократно использовались в этой главе (см. диаграммы на рис. 2.28–2.29). На рисунке 2.38 показаны деревья трех выражений.

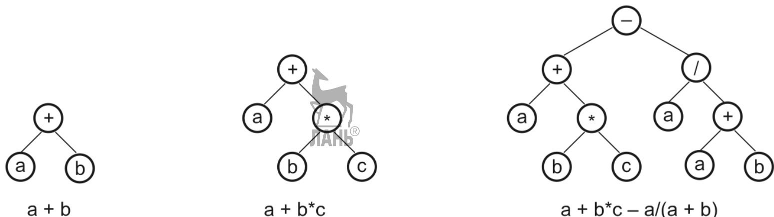


Рис. 2.38. Примеры семантических деревьев

Представление дерева

Для представления семантического дерева выражения будем использовать динамические переменные и указатели. Каждая вершина дерева содержит операнд или операцию. Поскольку в нашем примере используются однобуквенные операнды *a*, *b*, *c* и обозначения операций, состоящие из одного символа, достаточно предусмотреть в каждой записи, соответствующей вершине дерева, поле `Ch` символьного типа.

```
type
   tPtr = ^tNode;     {Тип указателя на вершину}
   tNode = record     {Тип вершины дерева      }
      Ch      : char;    {Символ вершины        }
      Counter: integer;{Счетчик временных переменных }
      Left    : tPtr;    {Указатель на левое поддерево }
      Right   : tPtr;    {Указатель на правое поддерево}
   end;
```

Поля `Left` и `Right` служат для установления связей в дереве. В нашем примере дерево будет двоичным, поскольку все операции двуместные. На поле `Counter` можно пока не обращать внимания,

оно потребуется позже для решения одной из задач. На рисунке 2.39 можно видеть дерево выражения $a + b*c$, состоящее из динамических переменных типа `tNode`.

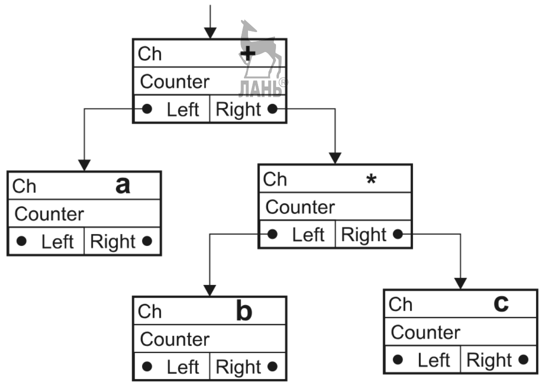


Рис. 2.39. Двоичное дерево выражения $a + b*c$

Построение дерева

Будем строить дерево с помощью программы, выполняющей синтаксический анализ выражения методом рекурсивного спуска. Потребуются следующие глобальные переменные:

```
var
   Ch     : char;    { Очередной символ            }
   Tree   : tPtr;    { Дерево                      }
   Counter: integer; { Счетчик временных переменных }
```

Задачей распознающих процедур `Expression` (выражение), `Term` (слагаемое) и `Multiplier` (множитель) кроме синтаксического анализа будет построение дерева того подвыражения, которое они распознали. `Expression` строит дерево выражения, `Term` — слагаемого, `Multiplier` — множителя. Построенное дерево будет выходным параметром каждой из этих процедур. Чтобы построить дерево `Tree` всего выражения обращаемся к `Expression`:

```
Expression(Tree);
```

`Expression` строит дерево выражения, обращаясь к распознавателю слагаемого для получения ссылок на деревья слагаемых. Связи

в дереве выстраиваются так, чтобы это соответствовало выполнению операций «+» и «–» в цепочке слагаемых в порядке слева направо.

```
procedure Expression(var Tree : tPtr);
var
   t : tPtr;
begin
   Term(Tree);
   while Ch in ['+', '-'] do begin
      New(t);       {Новая вершина}
      t^.Ch := Ch; {содержит знак}
      t^.Left := Tree; {Слева - старое дерево         }
      NextCh;
      Term(t^.Right);  {Справа - новое слагаемое     }
      Tree := t; {Указатель дерева - на новую вершину}
   end;
end;
```

Рисунок 2.40 иллюстрирует последовательность построения дерева для выражения $a - b - c$.

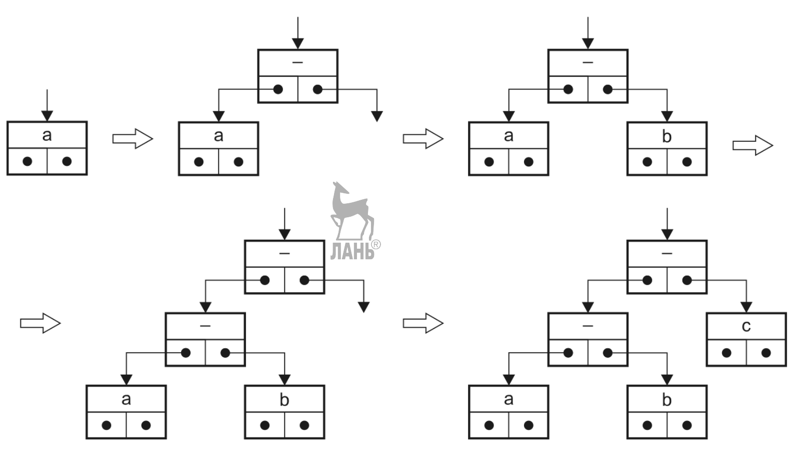


Рис. 2.40. Построение дерева выражения $a - b - c$

Анализатор слагаемого `Term`, действует подобно процедуре `Expression`.

```
procedure Term(var Tree : tPtr);
var
   t : tPtr;
```

```
begin
   Multiplier(Tree);
   while Ch in ['*', '/'] do begin
      New(t);
      t^.Ch := Ch;
      t^.Left := Tree;
      NextCh;
      Multiplier(t^.Right);
      Tree := t;
   end;
end;
```

Процедура-распознаватель множителя, встречая операнд, не заключенный в скобки, создает вершину, которая будет листом дерева.

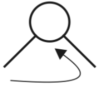
```
procedure Multiplier(var Tree : tPtr);
begin
   if Ch in ['a', 'b', 'c'] then begin
      New(Tree);
      Tree^.Ch := Ch;
      Tree^.Left := nil;
      Tree^.Right := nil;
      NextCh
      end
   else if Ch = '(' then begin
      NextCh;
      Expression(Tree);
      if Ch = ')' then
         NextCh
      else
         Error('Ожидается ")"');
      end
   else
      Error('Ожидается a, b, c или "("');
end;
```

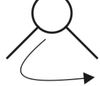
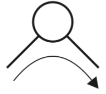
Дерево построено.

Получение постфиксной записи

Имея в распоряжении семантическое дерево, легко получить обратную польскую запись выражения. Для этого достаточно выполнить обход дерева (посещение всех его вершин) в порядке снизу вверх и слева направо, когда для каждой вершины вначале посещается ее левое поддерево, потом правое и в последнюю очередь сама

вершина (рис. 2.41*а*). Посещая каждую вершину, надо просто выводить ее содержимое (значение поля `Ch`).



а) Постфиксная запись　　б) Префиксная запись　　в) Инфиксная запись

Рис. 2.41. Различный порядок обхода дерева

Рекурсивный обход дерева в указанном порядке выполняет процедура `Postfix`.

```
{ Обход дерева t снизу вверх }
procedure Postfix(t: tPtr);
begin
   if t <> nil then begin
      Postfix(t^.Left);  { Обход левого поддерева  }
      Postfix(t^.Right); { Обход правого поддерева }
      Write(t^.Ch);      { Посещение вершины t     }
   end;
end;
```

Тривиальным случаем, когда не происходит рекурсивных вызовов (и не выполняется вообще никаких действий) здесь является ситуация, когда дерево пусто (`t = nil`).

Чтобы выполнить обход фактически и получить обратную польскую запись, нужно вызвать `Postfix`, указав фактическим параметром имеющееся дерево:

```
Postfix(Tree);
```

Получение префиксной записи

Префиксная (прямая польская) запись получается обходом дерева сверху вниз и слева направо.

```
{ Обход дерева t сверху вниз }
procedure Prefix(t: tPtr);
begin
   if t <> nil then begin
      Write(t^.Ch);
      Prefix(t^.Left);
      Prefix(t^.Right);
   end;
end;
```

Обход дерева `Tree` выполняется вызовом `Prefix(Tree)`. В таблице 2.11 приведен результат работы процедур `Postfix` и `Prefix` для трех выражений, чьи деревья можно видеть на рисунке 2.38.

Таблица 2.11. Результат работы процедур Postfix и Prefix

Выражение	Postfix	Prefix
a + b	ab+	+ab
a + b*c	abc*+	+a*bc
a + b*c – a/(a + b)	abc*+aab+/–	–+a*bc/a+ab

Функциональная запись выражения

Согласитесь, что если обратная польская запись стала уже привычной, то префиксная воспринимается пока с трудом. Большую наглядность ей можно придать, если считать, что каждая операция — это функция с двумя аргументами. Обозначение функции записывается перед обозначением аргументов: $f(x, y)$. Заменив знаки операций названиями функций: ADD — сложение, SUB — вычитание, MUL — умножение, DIV — деление, а также предусмотрев вывод запятых и скобок, можно получить запись выражения в функциональном стиле. При этом она, по сути, остается префиксной, лишь несколько меняя внешний вид.

При посещении вершин дерева надо различать, является ли вершина внутренней или листом. Для вершин-операций надо выводить название функции, скобки и запятую между аргументами. Попав в вершину-операнд, достаточно вывести символ, хранящийся в этой вершине, и даже не нужно посещать поддеревья, поскольку такая вершина их не имеет, являясь листом дерева.

Чтобы различать операции и операнды, используем множество

```
Operators = ['+', '-', '*', '/'].
```

Обход дерева по-прежнему надо выполнять сверху вниз и слева направо.

```
{ Функциональная запись дерева t }
procedure Functional(t: tPtr);
begin
   if t <> nil then begin
      if t^.Ch in Operators then begin
         WriteMnemo(t^.Ch); { Название функции }
         Write('(');
         Functional(t^.Left);
         Write(', ');
```

```
            Functional(t^.Right);
            Write(')');
            end
         else { Лист }
            Write(t^.Ch);
      end;
   end;
```

Вывод названия функции, соответствующего знаку операции, выполняет такая процедура:

```
procedure WriteMnemo(Op: char);
begin
   case Op of
   '+': Write('ADD');
   '-': Write('SUB');
   '*': Write('MUL');
   '/': Write('DIV');
   end;
end;
```

Чтобы получить функциональную запись выражения, для которого построено дерево `Tree`, надо написать: `Functional(Tree)`. Результат такого вызова для трех случаев показан в таблице 2.12.

Таблица 2.12. Функциональная запись выражений

Выражение	Функциональная запись
`a + b`	`ADD(a, b)`
`a + b*c`	`ADD(a, MUL(b, c))`
`a+b*c-a/(a+b)`	`SUB(ADD(a, MUL(b, c)), DIV(a, ADD(a, b)))`

Четверки

Четверки — одна из форм промежуточного представления программы в трансляторе. Представление четверками удобно для выполнения оптимизирующих преобразований программы, увеличивающих скорость ее работы и уменьшающих размер. Поэтому четверки используются в оптимизирующих компиляторах.

Четверку можно рассматривать как команду некоторого компьютера, имеющую следующий формат:

```
Операция, Операнд-1, Операнд-2, Результат
```

Например, выражение $a + b$ может быть преобразовано в четверку

```
+, a, b, t
```

Здесь t обозначает временную переменную, в которую помещается результат сложения.

Некоторые реальные компьютеры действительно имели и имеют аналогичную (трехадресную) систему команд, когда за кодом операции в команде следуют адреса двух операндов и адрес результата. Интересно заметить, что это либо старые машины 60-х годов (такие, как советская М-20), либо, наоборот, современные процессоры с сокращенным набором команд (RISC). Только если в старых трехадресных машинах в команде указывались адреса ячеек памяти, то в командах RISC-процессоров — номера регистров.

Не надо, однако, думать, что порождение четверок имеет смысл только при компиляции для трехадресных машин. Четверки — универсальный машинно-независимый способ промежуточного представления программы, по которому затем может быть получен машинный код для компьютеров различной архитектуры.

Одна операция в арифметическом выражении порождает одну четверку. Выражение в целом — последовательность четверок. В наших примерах в роли результата всегда будут фигурировать временные переменные, которые будем обозначать $t1$, $t2$, и т. д. (от temporary — временный). Номер четверки будет и номером временной переменной. Результат вычисления выражения сохраняется в последней временной переменной.

Получить четверки можно обходом семантического дерева снизу вверх. Последняя четверка соответствует последней выполняемой операции, которая находится в корне дерева. Посещать при обходе нужно только внутренние вершины дерева, соответствующие операциям.

Для назначения номеров временным переменным и, соответственно, четверкам мы уже предусмотрели переменную `Counter` и поле `Counter` записи о вершине дерева. При выводе очередной четверки значение `Counter` увеличивается на единицу (первоначально `Counter := 0`) и используется для нумерации временной переменной, являющейся результатом в этой четверке. Это же значение записывается в вершину дерева, для которой выводилась четверка, становясь номером этой вершины.

Если левой (правой) дочерней вершиной той внутренней вершины, для которой выводится четверка, является элементарный операнд (a, b или c), его обозначение выводится в качестве первого (второго)

операнда четверки. Если дочерняя вершина внутренняя, выводится «t» и номер дочерней вершины, записанный ранее в ее поле Counter.

```
{ Обход дерева t для получения четверок }
procedure Tetrads(t: tPtr);
begin
  if (t <> nil) and (t^.Ch in Operators) then begin
    Tetrads(t^.Left); {Четверки для левого поддерева }
    Tetrads(t^.Right);{Четверки для правого поддерева}
    { Четверка для вершины t }
      Write{Mnemo}(t^.Ch); { Операция }
      Operand(t^.Left);    { Первый операнд }
      Operand(t^.Right);   { Второй операнд }
      { Результат }
        Counter := Counter + 1;
        WriteLn(', t', Counter);
    { Пронумеровать вершину }
      t^.Counter := Counter;
  end;
end;
```

Вывод операндов выполняется процедурой Operand:

```
procedure Operand(p : tPtr);
begin
   if p^.Ch in Operators then
      Write(', t', p^.Counter)
   else
      Write(',  ', p^.Ch);
end;
```

Таблица 2.13. Четверки

Выражение	Четверки	Четверки с мнемоникой
a + b	+, a, b, t1	ADD, a, b, t1
a + b*c	*, b, c, t1 +, a, t1, t2	MUL, b, c, t1 ADD, a, t1, t2
a + b*c – a/(a + b)	*, b, c, t1 +, a, t1, t2 +, a, b, t3 /, a, t3, t4 -, t2, t4, t5	MUL, b, c, t1 ADD, a, t1, t2 ADD, a, b, t3 DIV, a, t3, t4 SUB, t2, t4, t5

Получить четверки для выражения, по которому построено дерево `Tree`, можно вызовом `Tetrads(Tree)`. Последовательность четверок для выражения, состоящего из единственного операнда, будет пуста.

Заменив `Write(t^.Ch)` на `WriteMnemo(t^.Ch)`, можно получить четверки с мнемоническими обозначениями операций (см. табл. 2.13). Такие четверки еще больше похожи на команды машинного языка.

Вычисление (интерпретация) выражения по его семантическому дереву

Выполнив обход дерева снизу вверх, как при формировании обратной польской записи и четверок, можно вычислить значение выражения. Надо только уметь извлекать значения элементарных операндов. Вообще-то, *a*, *b* и *c*, исполняющие эту роль в наших примерах, могут быть переменными или константами, их значения могут считываться из ячейки памяти или, например, с какого-либо прибора, подключенного к компьютеру. В зависимости от этого и способ получения их значений будет различным. Но, чтобы не усложнять изложение, примем, что *a* просто равно 1, *b* равно 2, а $c = 3$.

Процедуру `Calculate`, выполняющую обход дерева с целью вычисления значения выражения, снабдим выходным параметром `x`, обозначающим найденное значение. Будем считать, что это значение вещественное.

```
{ Обход дерева t для вычисления выражения }
procedure Calculate(t : tPtr; var x : real);
var
   x1, x2 : real;
begin
   if t<>nil then begin
      { Вычислить в левом поддереве }
         Calculate(t^.Left, x1);
      { Вычислить в правом поддереве }
         Calculate(t^.Right, x2);
      { Найти значение в вершине t }
         case t^.Ch of
         '+':  x := x1 + x2;
         '-':  x := x1 - x2;
         '*':  x := x1*x2;
         '/':  x := x1/x2;
         'a':  x := 1.0; { Так          }
         'b':  x := 2.0; { только       }
```

```
         'c':  x := 3.0; { для примера }
         end;
   end;
end;
```

Чтобы, допустим, напечатать значение выражения, для которого построено дерево Tree, можно записать:

```
var
   y : real;
…
Calculate(Tree, y);
Writeln('Значение равно ', y);
```

Поскольку результатом работы Calculate является единственное значение, имеет смысл оформить вычисление выражения по его семантическому дереву как функцию. Назовем ее Value — «значение».

```
{ Значение выражения, заданного деревом t }
function Value(t : tPtr): real;
begin
   if t<>nil then begin
      case t^.Ch of
      '+':  Value := Value(t^.Left) + Value(t^.Right);
      '-':  Value := Value(t^.Left) - Value(t^.Right);
      '*':  Value := Value(t^.Left) * Value(t^.Right);
      '/':  Value := Value(t^.Left) / Value(t^.Right);
      'a':  Value := 1.0;
      'b':  Value := 2.0;
      'c':  Value := 3.0;
      end;
   end;
end;
```

Согласитесь, получилось изящно. Теперь для печати значения достаточно написать:

```
Writeln('Значение равно ', Value(Tree));
```

Результат вычисления нескольких выражений с помощью Calculate и Value приведен в таблице 2.14.

Хочется отметить важный нюанс. Программы, построенные по образцу Calculate с одной стороны и Value — с другой, могут оказаться неэквивалентными. Дело в том, что в некоторых языках программирования не фиксирован порядок вычисления операндов арифметической операции.

Таблица 2.14. Значения выражений, вычисленные обходом дерева

Выражение	Значение
`a`	`1.000`
`b`	`2.000`
`c`	`3.000`
`a + b`	`3.000`
`a + b*c`	`7.000`
`a + b*c - a/(a + b)`	`6.667`

Запись `Value(t^.Left) + Value(t^.Right)` совсем не обязательно означает, что вычисление значения в левом поддереве будет выполнено раньше вычисления в правом. Если операндами выражения являются функции с побочным эффектом, то в зависимости от порядка вычисления операндов значение выражения может оказаться различным. А вот программа подобная `Calculate` гарантирует, что вначале посещается левое поддерево и лишь потом — правое.

Получается, что `Value` будет всегда вычислять левый операнд раньше правого только в том случае, если это гарантирует реализация того языка, на котором сама функция `Value` (или подобная ей) написана.

Упражнения для самостоятельной работы

Ниже приводятся описания ряда конструкций (выражений, уравнений, элементов программ) и варианты их обработки. В качестве упражнений предлагается:

- Построить синтаксические диаграммы для этих конструкций.
- Запрограммировать синтаксический анализатор методом рекурсивного спуска. Анализатор должен либо сообщать о том, что конструкция записана верно, либо выдавать сообщение об ошибке с указанием места ее обнаружения.
- Построить синтаксическую таблицу и реализовать табличный $LL(1)$-анализатор.
- Дополнить анализаторы семантическими процедурами, выполняющими содержательную обработку.

Варианты заданий

1. Сумма — последовательность натуральных чисел и имен, разделенных знаками плюс и минус. Возможен и знак перед первым слагаемым.
2. Сумма вещественных чисел.
3. Квадратное уравнение с целыми коэффициентами.
4. Линейное алгебраическое уравнение с N неизвестными (X_k, $k = 1,2 \ldots N$) и постоянными целыми коэффициентами.
5. Сумма обыкновенных дробей.
6. Комплексное число (с целочисленными значениями действительной и мнимой частей).
7. Линейное однородное дифференциальное уравнение с постоянными целочисленными коэффициентами.
8. Произведение вида $(X - A_1)(X - A_2)(X - A_3) \ldots (X - A_n)$, где A_i, ($i = 1 \ldots n$) — целые числа.
9. Линейное уравнение с n неизвестными и постоянными целочисленными коэффициентами. Имена переменных произвольные.
10. Мультипликативная функция n переменных: $\Phi = A_0 \prod_{i=1}^{n} X_i$, где A_0 — целая константа; X_i, ($i = 1 \ldots n$) — имена.
11. Отношение *Операнд* ⊗ *Операнд*, где *Операнд* – целое или имя; ⊗ – знак отношения (>, <, =, <>, >=, <=).
12. Условие — отношения (см. выше), объединенные операциями «**и**» и «**или**». Операнды — однобуквенные имена.
13. Сумма произведений – последовательность слагаемых, представляющих собой произведение нескольких операндов. Операнды — имена.
14. Последовательность записанных через запятую элементов двумерных массивов. Индексы – натуральные числа.
15. Серия команд присваивания, разделенных «;». В каждой команде слева записано имя, справа — натуральное число. Знак присваивания «:=».

16. Сумма функций — последовательность функций с произвольными именами, разделенных знаками «+» и «–». Аргументы функций — имена или натуральные числа в скобках.

17. Бесскобочное арифметическое выражение с функциями. Все операнды — однобуквенные имена. Аргументы функций записываются в скобках.

18. Оператор `Write` (элементы списка — имена, строки; параметры форматов — целые числа).

19. Оператор `Read` с элементами списка ввода — именами простых переменных и элементами линейных массивов. Индексы массивов — целые числа.

20. Переменная языка Паскаль (включая элементы массива, компоненты записей и динамические переменные). Индексы — целые числа.

21. Простой тип языка Паскаль (имя, перечислимый тип, ограниченный тип). Константы ограниченного типа — целые числа или символы.

22. Раздел констант языка Паскаль. Предполагаются константы только целого и символьного типа.

23. Числовой ряд вида $1 + 1/n_1 + 1/n_2 + \dots$, где $n_1, n_2, \dots$ — натуральные числа.

24. Числовой ряд вида $1 + 1/(n_1 * m_1) + 1/(n_2 * m_2) + \dots$, где $n_1, n_2, \dots, m_1, m_2, \dots$ — натуральные числа.

25. Числовой ряд вида $1 + 1/n_1$^2 $+ 1/n_2$^2 $+ \dots$, где $n_1, n_2, \dots$ — натуральные числа.

26. Многочлен от x с рациональными коэффициентами.

27. Дробно-линейная функция от x: $(a_1 * x + b_1)/(a_2 * x + b_2)$, где a_1, b_1, a_2, b_2 — целые числа (при равенстве 0 могут не записываться).

28. Иррациональная сумма вида x^$(n_1/m_1) + x$^$(n_2/m_2) - \dots + \dots$, где $n_1, n_2, \dots, m_1, m_2, \dots$ — натуральные числа.

29. Последовательность векторов: (x_1, y_1, z_1), (x_2, y_2, z_2), ..., где x, y, z — целочисленные координаты векторов.

30. Многочлен от x: $(x - x_1)$^$k_1 * (x - x_2)$^$k_2 * \dots$, где $x_1, x_2, \dots$ — целые числа; $k_1, k_2, \dots$ — неотрицательные целые.

31. Многочлен от x, записанный по схеме Горнера: $a_0 + x(a_1 + x(a_2 + x(\ldots)))$, где a_0, a_1, ... — целые числа.

32. Дробно-рациональная функция: $((x - a_1)(x - a_2) \ldots)/((x - b_1)(x - b_2) \ldots)$, где $a_1, a_2, \ldots, b_1, b_2, \ldots$ — целые числа.

33. Уравнение плоскости вида $Ax + By + Cz + D = 0$ с целыми коэффициентами.

34. Уравнение плоскости в отрезках: $x/a + y/b + z/c = 1$, где a, b, c — ненулевые целые числа.

35. Линейная функция двух переменных (x и y) с вещественными коэффициентами.

36. Множество-константа языка Паскаль с базовым типом `char`.

37. Множество-константа языка Паскаль с базовым типом `integer`.

Варианты обработки

1. Считая, что имена обозначают неизвестные, запросить у пользователя их значения и вычислить сумму. Значение одной и той же переменной, встречающейся в выражении больше одного раза, должно запрашиваться только один раз.

2. Вычислить сумму.

3. Найти корни уравнения.

4. Программа обеспечивает ввод числовых значений неизвестных и проверяет, являются ли эти значения решениями уравнения. Вводятся только значения неизвестных, фактически имеющихся в уравнении. Если некоторая неизвестная встречается дважды, программа должна запрашивать ее значение только один раз.

5. Вычислить сумму.

6. Вычисляется z^2, z^3, z^4, ..., z^{10}, где z — введенное комплексное число.

7. Вычисляются корни характеристического уравнения.

8. Программа запрашивает значение x и вычисляет произведение.

9. Программа обеспечивает ввод числовых значений неизвестных и проверяет, являются ли эти значения решениями уравнения. Вводятся только значения неизвестных, фактически имеющихся в уравнении. Если некоторая неизвестная встречается дважды, программа должна запрашивать ее значение только один раз.

10. Программа запрашивает вещественные значения входящих в выражение переменных и вычисляет значение функции. Значение каждой переменной вводится лишь однажды.
11. Полагая, что имена — это идентификаторы целочисленных переменных, программа запрашивает их значения и проверяет истинность отношения.
12. Полагая, что имена суть идентификаторы целочисленных переменных, программа запрашивает их значения и проверяет истинность условия. Использовать короткую схему вычисления логических выражений, не требуя ввода значений переменных, если истинность условия может быть определена без них.
13. Программа запрашивает вещественные значения, входящих в выражение переменных и вычисляет значение суммы. Значение каждой переменной вводится лишь однажды.
14. Считая, что все массивы имеют размер 10×10, программа проверяет корректность задания индексов, и формирует текст программы на Паскале, которая заполняет все массивы, имена которых встретились в выражении, случайными целыми числами, элементам, встретившимся в выражении должны быть присвоены нулевые значения. Сгенерированная программа должна распечатывать содержимое всех этих массивов.
15. Сгенерировать программу на Паскале, состоящую из этих присваиваний и оператора `Write`, печатающего значения этих переменных.
16. Сгенерировать синтаксически правильную (не обязательно осмысленную) программу на Паскале, частью которой является оператор присваивания, содержащий в правой части исходное выражение.
17. Спрашивая у пользователя вещественное значение каждой переменной и значение каждой функции при заданном числовом значении аргумента, программа-интерпретатор вычисляет выражение.
18. Выполнить оператор, запросив вещественные значения переменных.
19. Сгенерировать синтаксически правильную программу на Паскале, первым в которой записан данный оператор.

20. Сгенерировать синтаксически правильную программу на Паскале, содержащую описание этой переменной.

21. Сгенерировать программу на Паскале, которая содержит цикл, параметр которого меняется от минимального до максимального значения данного типа. Если тип задан именем, то предполагается, что это имя одного из стандартных дискретных типов. Программа должна проверить это.

22. Сгенерировать программу на Паскале, в которой данные константы определены как типизированные.

23. Вычислить точную сумму ряда. Результат представить в виде обыкновенной дроби.

24. Вычислить точную сумму ряда. Результат представить в виде обыкновенной дроби.

25. Вычислить точную сумму ряда. Результат представить в виде обыкновенной дроби.

26. Программа запрашивает рациональное значение x, а затем вычисляет значение многочлена, представив ответ в виде несократимой дроби.

27. Запросив целое значение x, вычислить точное значение функции, представив его в виде обыкновенной дроби.

28. Запросив вещественное значение x, вычислить сумму.

29. Вычислить длину ломаной, рассматривая данные векторы как радиус-векторы узлов ломаной в трехмерном пространстве.

30. Вычисляется многочлен при заданном x.

31. Вычисляется многочлен при заданном x.

32. Запросить рациональное значение x. Вычислить значение функции при заданном x, представив это значение в виде обыкновенной дроби.

33. Вычисляются коэффициенты плоскости, перпендикулярной заданной и проходящей через ось Ox.

34. Если плоскость не параллельна ни одной из координатных осей, вычислить объем тетраэдра, образованного заданной плоскостью и координатными плоскостями. Если параллельна, сообщить об этом.

35. Программа запрашивает значения x и y и вычисляет значение функции.
36. Программа запрашивает символ и отвечает на вопрос о принадлежности этого символа заданному множеству.
37. Программа запрашивает целое число и отвечает на вопрос о принадлежности этого числа заданному множеству.

Глава 3. Трансляция языков программирования

Вооружившись теорией трансляции, перейдем к рассмотрению методов конструирования компиляторов. Будем рассматривать вопросы разработки транслятора на конкретном примере. Большая часть этой главы посвящена созданию компилятора (он же окажется интерпретатором) простого языка программирования.

Описание языков программирования

Нельзя говорить о разработке транслятора, если нет точного и строгого описания языка, для которого этот транслятор изготавливается. Конечно при создании новых языков ситуация не всегда так определённа. Документ, определяющий язык — его спецификация, — бывает, пишется одновременно с разработкой компилятора. В ходе реализации (создания транслятора) в язык могут вноситься изменения и уточнения. Но рано или поздно спецификация должна быть сформулирована. Для многих распространенных языков существуют международные стандарты. И, если речь идет об уже существующем языке, без его спецификации разработчику компилятора обойтись невозможно.

Важно подчеркнуть, что не любое описание языка годится на роль документа, руководствуясь которым можно программировать компилятор. О различных языках программирования написано множество книг. Это и учебники, и «наиболее полные руководства», и краткие справочники, и пособия «для чайников», а также краткие и вводные курсы. Но ни одна из таких книг не годится в качестве описания языка при создании компилятора. Эти книги, конечно, нужны для знакомства с языком, лучшего его понимания, как источники примеров. Но разработчику компилятора требуется спецификация. Это может быть стандарт, если он существует, либо авторское описание языка. Спецификация — это максимально точное определение языка, его синтаксиса и семантики. Спецификация — весьма строгий и сухой документ, который не может служить учебником по программированию. Она обычно содержит не слишком много примеров, не дает рекомендаций по применению тех или иных конструкций, а лишь определяет их форму и смысл.

Метаязыки

Описание языка должно быть тоже записано на некотором языке, который выступает в этом случае в роли *метаязыка*. Для описания синтаксиса используются формальные нотации, эквивалентные порождающим грамматикам Н. Хомского. Синтаксис языков программирования обычно определяется с помощью контекстно-свободных грамматик, быть может, с несколько расширенными правилами.

Несмотря на то, что существуют подходы к формализации описания семантики (например, атрибутные грамматики), они не получили распространения в практике спецификации языков программирования. Семантику языковых конструкций определяют, пользуясь естественным языком — английским, русским.

Ниже мы рассмотрим варианты нотаций, использованных при описании синтаксиса известных языков программирования.

БНФ

Бэкуса — Наура форма (БНФ) была впервые применена при описании Алгола-60. БНФ совпадает по сути с нотацией КС-грамматик, отличаясь лишь обозначениями. Предусмотрены следующие метасимволы:

< > — служат для выделения нетерминалов — понятий языка.

| — «или». Разделяет альтернативные правые части правил.

::= — «есть по определению». Заменяет стрелку, используемую при записи продукций КС-грамматик.

Терминальные символы записываются как есть, никаких специальных способов их выделения не предусмотрено.

Вот пример определений на БНФ, взятый из спецификации Алгола-60 — «Модифицированного сообщения»:

```
<простое арифметическое выражение> ::=
   <терм> | <знак операции типа сложения> <терм> |
   <простое арифметическое выражение>
   <знак операции типа сложения> <терм>
<знак операции типа сложения> ::= + | -
```

Как видно, для выражения повторений используется рекурсия, причем повсеместно — левая.

БНФ использована Н. Виртом при описании языка Паскаль. Хотя в нотацию были добавлены метаскобки { и }, обозначающие повто-

рение, применены они лишь в отдельных случаях, в то время как, например, грамматика выражений леворекурсивна.

Синтаксические диаграммы

Нет особой необходимости знакомить читателя с синтаксическими диаграммами, поскольку они уже рассматривались в этой книге. Диаграммы стали популярны после выхода книги К. Йенсен и Н. Вирта «Паскаль». Они использованы в первой ее части — Руководстве — компактном учебнике языка. На рисунке 3.1 показана одна из имеющихся там диаграмм.

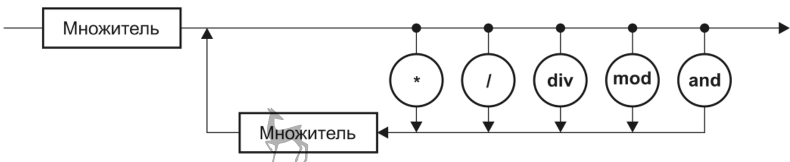


Рис. 3.1. Синтаксическая диаграмма слагаемого (язык Паскаль)

Расширенная форма Бэкуса — Наура (РБНФ)

Как уже говорилось, отсутствие в нотации формальных грамматик (и БНФ) средств явного задания повторений создает ряд трудностей. Во-первых, определения оказываются сложными для понимания, недостаточно наглядными из-за обилия рекурсий. Во-вторых, возникают проблемы с тем, что грамматики, дающие подходящие семантические деревья, оказываются леворекурсивными.

При описании Модулы-2 и Оберона Н. Вирт использовал расширенную Бэкуса — Наура форму (РБНФ)[58]. Главные модификации касаются введения скобок { и } для повторений, а [и] — для обозначения необязательного вхождения цепочек терминалов и нетерминалов в правые части правил. Соглашения относительно обозначений терминалов и нетерминалов также изменены, что не столь принципиально.

В дальнейшем мы будем пользоваться именно РБНФ. Вот как она определяется в спецификации Оберона-2[59]:

Варианты разделяются знаком |. Квадратные скобки [и] означают необязательность записанного внутри них выражения, а фигурные скобки { и } означают его повторение (возможно, 0 раз). Нетерми-

---

[58] Эта нотация была предложена им в 1977 году [Вирт, 1977].

[59] Точнее, в русском переводе спецификации.

нальные символы начинаются с заглавной буквы (например, `Оператор`). Терминальные символы или начинаются строчной буквой (например, `идент`), или записываются целиком заглавными буквами (например, `BEGIN`), или заключаются в кавычки (например, `":="`).

К этому следует добавить, что в роли знака «есть по определению» в РБНФ используется «=», а каждое правило заканчивается точкой.

Вот так может быть определен синтаксис идентификатора (имени) с помощью РБНФ:

```
Имя = Буква { Буква | Цифра }.
```

Являясь метаязыком, РБНФ должна быть пригодна для описания языков, имеющих практический интерес. В том числе с помощью РБНФ может быть определен и синтаксис самой РБНФ[60]:

```
Синтаксис = { Правило }.
Правило   = Имя "=" Выражение ".".
Выражение = Вариант { "|" Вариант }.
Вариант   = Элемент { Элемент }.
Элемент   = Имя | Цепочка | "(" Выражение ")" |
            "[" Выражение "]" | "{" Выражение "}".
Цепочка   = "'" { символ } "'" | '"' { символ } '"'.
```

В этих определениях не сделано различий между именами, обозначающими терминалы и нетерминалы, хотя сформулировать это на РБНФ было бы несложно. Различение имён вынесено за рамки синтаксиса и может быть специфицировано (и специфицируется, см. выше) отдельно. Подобным же образом часто поступают при определении языков программирования.

Описания синтаксиса языков семейства Си

В знаменитой книге Б. Кернигана и Д. Ритчи описание синтаксиса языка Си дано в нотации, которая эквивалентна БНФ, но использует другие соглашения об обозначениях терминалов и нетерминалов, альтернативных правых частей правил, необязательных конструкций.

Нетерминалы записываются курсивом, терминалы — прямым шрифтом. Альтернативные части правил выписываются в столбик по одному в строке или помечаются словами «one of» (один из). Необязательные части сопровождаются подстрочным индексом $_{opt}$ (от op-

---

[60] Подобное, конечно, справедливо и в отношении БНФ. РБНФ, как БНФ и как рассмотренный выше язык регулярных выражений, — это КС-язык.

tional — необязательный; *необ.* — в некоторых русских переводах). Левая часть правила записывается отдельной строкой с отступом влево.

Вот пример определений конструкций языка Си:

составной оператор
  { *список описаний*$_{opt}$ *список операторов*$_{opt}$ }
список операторов
  оператор
  оператор список операторов

Как видно, из-за отсутствия явного способа выражения повторений, определения изобилуют рекурсией.

Аналогичная нотация с минимальными изменениями использована для описания синтаксиса языков-потомков Си: Си++, Ява, Си#. Вот выдержка из стандарта ECMA-334 на язык Си#:

block:
  { *statement-list*$_{opt}$ }
statement-list:
  statement
  statement-list statement

Специального названия эта нотация, судя по всему, не имеет. И представляется, как минимум, странной. Она неудобна не только для чтения, но и для обработки на компьютере. Ее использование при описании новых языков трудно объяснить чем-либо, кроме дурно понятой необходимости следовать традициям.

Описания синтаксиса языка Ада

Контекстно-свободные грамматики языков Ада-83 и Ада-95 определены с помощью варианта БНФ, в который добавлены обозначения повторений и необязательных частей. Названия нетерминалов записываются обычным шрифтом с использованием знака подчеркивания, если название составное, а зарезервированные слова — жирным шрифтом. Поскольку ни квадратные, ни фигурные скобки в Аде не используются, как не используется и знак «|» (все это метасимволы), никакого специального обозначения для терминалов не предусмотрено.

Определение синтаксиса оператора **if**, взятое из стандарта Ада-95, выглядит так:

```
if_statement ::=
   if condition then
```

```
      sequence_of_statements
   {elsif condition then
      sequence_of_statements}
   [else
      sequence_of_statements]
   end if;
```

Интересная и полезная особенность: синтаксические правила структурных конструкций представлены в виде, соответствующем их рекомендованному форматированию в программах (разделение на строки, отступы).

Язык программирования «О»

Все последующие вопросы, связанные с конструированием компиляторов, мы будем рассматривать на примере разработки транслятора для минимального подмножества языка Оберон-2. Такой конкретный и предметный подход наверняка сузит круг обсуждаемых тем, не позволит рассмотреть тонкие и сложные вопросы, связанные с оптимизацией кода, зато гарантирует полное понимание основ.

Тот простой язык, для которого мы напишем компилятор, будет называться «О». Это шуточное название, во-первых, подчеркивает некоторую несерьезность, учебный характер, «игрушечность» языка — уж очень он будет прост. Во-вторых, буква «О», похожа на ноль (сложность языка стремится к нулю) и в то же время это первая буква слова Оберон. Из других ассоциаций — «Операция «Ы».

Краткая характеристика языка «О»

- Язык «О» является точным подмножеством Оберона и Оберона-2, то есть любая программа на языке «О» является правильной программой на языках Оберон и Оберон-2.
- Программа на языке «О» состоит из единственного модуля. Выполнение программы начинается с первого оператора, записанного после слова **BEGIN**. Процедуры в языке «О» отсутствуют.
- Предусмотрены константы и переменные только целого (INTEGER) типа. Выражения логического типа (без логических операций) могут использоваться в операторах **IF** и **WHILE**. Массивов и записей нет.
- Выражения строятся по правилам языка Оберон. Допустимы все операции, применимые к целым и дающие результат целого типа: +, –, *, **DIV**, **MOD**. В логических выражениях используются операции

отношения: =, #, <, >, <=, >=, которые применимы к целочисленным операндам.

- Набор операторов включает присваивание, вызов процедуры (стандартной), **IF** – **THEN** – **ELSIF** ... **ELSE** – **END**, **WHILE** – **DO** – **END**.
- Предусмотрены стандартные процедуры и процедуры-функции `ABS`, `DEC`, `HALT`, `INC`, `MAX`, `MIN`, `ODD`. Их смысл такой же, как и в языке Оберон-2.
- Разрешается импорт стандартных (псевдо) модулей `In` и `Out`, предоставляющих процедуры ввода-вывода `In.Open`, `In.Int`, `Out.Int`, `Out.Ln`. Их свойства приведены в таблице.

Название процедуры	**Параметры**	**Описание**
`In.Open`	Нет	Открывает стандартный входной поток
`In.Int(v)`	`v: INTEGER`	Вводит значение `v`
`Out.Int(x, n)`	`x, n: INTEGER`	Выводит значение `x`, используя `n` позиций
`Out.Ln`	Нет	Выводит перевод строки

- В записи программы разрешены комментарии, которые могут быть вложенными.
- Большие и малые буквы различаются.
- Кроме ключевых слов, используемых в языке «О», зарезервированными считаются также все остальные служебные слова языка Оберон-2. Их нельзя использовать в качестве идентификаторов в программах на «О». Это обеспечивает полную совместимость снизу вверх с Обероном-2.

Синтаксис «О»

Ниже приводится сводка синтаксиса языка «О» на РБНФ.

```
Модуль =
   MODULE Имя ";"
   [Импорт]
   ПослОбъявл
   [BEGIN
      ПослОператоров]
   END Имя ".".
Импорт =
   IMPORT Имя {"," Имя} ";".
ПослОбъявл =
```

```
   {CONST
      {ОбъявлКонст ";"}
   |VAR
      {ОбъявлПерем ";"} }.
ОбъявлКонст = Имя "=" КонстВыраж.
ОбъявлПерем = Имя {"," Имя} ":" Тип.
Тип = Имя.
ПослОператоров =
   Оператор {";"
   Оператор }.
Оператор = [
   Переменная ":=" Выраж
   |[Имя "."] Имя ["(" [Параметр {"," Параметр}] ")"]
   |IF Выраж THEN
      ПослОператоров
   {ELSIF Выраж THEN
      ПослОператоров}
   [ELSE
      ПослОператоров]
    END
   |WHILE Выраж DO
      ПослОператоров
    END
].
Параметр = Переменная | Выраж.
Переменная = Имя.
КонстВыраж = ["+" | "-"] (Число | Имя).
Выраж = ПростоеВыраж [Отношение ПростоеВыраж].
ПростоеВыраж = ["+"|"-"] Слагаемое {ОперСлож Слагае-
мое}.
Слагаемое = Множитель {ОперУмн Множитель}.
Множитель =
   Имя ["(" Выраж | Тип ")"]
   | Число
   | "(" Выраж ")".
Отношение = "=" | "#" | "<" | "<=" | ">" | ">=".
ОперСлож = "+" | "-".
ОперУмн = "*" | DIV | MOD.
Имя = буква {буква | цифра}.
Число = цифра {цифра}.
```

Можно заметить, что грамматика языка «О» не является *LL*(1)-грамматикой. Действительно, две первых альтернативы правила для операторов (первая соответствует присваиванию, вторая — вызову

процедуры) нельзя различить по одному очередному символу — переменная представляет собой имя. Оба варианта в списке фактических параметров процедур и функций (правила для Оператора и Множителя) также могут начинаться именем. Можно было бы выполнить преобразование грамматики, но в этом нет особой нужды. Разделение этих случаев легко выполняется с помощью несложных контекстных проверок: если имя, начинающее оператор, обозначает переменную, то впереди присваивание, иначе предполагаем вызов процедуры. Варианты анализа фактических параметров можно выбрать по виду соответствующих формальных.

Пример программы на «О»

Программа `Primes` печатает простые числа в диапазоне от 2 до *n* и подсчитывает их количество.

```
MODULE Primes;
(* Простые числа от 2 до n *)

IMPORT In, Out;

VAR
   n, c, i, d : INTEGER;

BEGIN
   In.Open;
   In.Int(n);
   c := 0; (* Счетчик простых *)
   i := 2;
   WHILE i <= n DO
      (* Делим на 2, 3, ... пока не разделится *)
         d := 2;
         WHILE i MOD d # 0 DO
            INC(d)
         END;
      IF d = i THEN (* i - простое *)
         INC(c);
         Out.Int(d, 8)
      END;
      INC(i);
   END;
   Out.Ln;
   Out.Int(c, 0);
END Primes.
```

Использованный здесь алгоритм весьма неэффективен[61], зато очень прост[62].

Структура компилятора

Программы-компиляторы могут во многом отличаться друг от друга по внутреннему устройству, но есть несколько частей, которые присутствуют всегда или почти всегда. На рисунке 3.2 показана типовая структура компилятора.

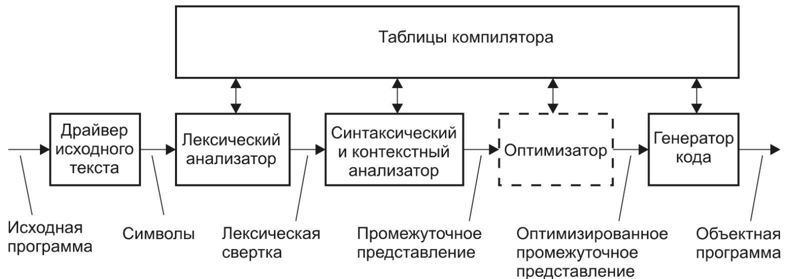


Рис. 3.2. Структура компилятора

Блок оптимизатора, показанный пунктиром, не является обязательным.

В дальнейшем мы подробно рассмотрим и запрограммируем модули компилятора, а сейчас — короткие пояснения.

Драйвер исходного текста. Непосредственно взаимодействует с текстом программы. Необходимость в этом модуле обусловлена тем, что источник текста бывает разным: программа может вводиться из файла или извлекаться из базы данных, считываться из окна текстового редактора, загружаться по сети и т. д. Драйвер может буферизовать ввод, обеспечивать вывод исходного текста по ходу трансляции, вести счет строк и символов, обеспечивать показ фрагмента программы, вызвавшего ошибку. Следующим модулям компилятора драйвер передает последовательность символов. При этом остальные части транслятора не зависят от источника и способа представления исходного текста. Основную функцию драйвера выполняет процедура `NextCh`.

[61] Быть может, это самый неэффективный алгоритм поиска простых чисел.

[62] Возможно даже, что это самый простой алгоритм решения этой задачи.

Лексический анализатор (сканер). Разбивает программу на простые элементы-слова, называемые *лексемами*. Лексемами являются имена, служебные слова, числа, разделители, знаки операций. Удаляет из программы комментарии. Использование лексического анализатора упрощает последующие модули, которые уже не должны иметь дела с отдельными знаками, а могут оперировать более крупными неделимыми единицами.

Синтаксический и контекстный анализаторы. Суть работы синтаксического анализатора нам хорошо знакома. Синтаксического анализа недостаточно для того, чтоб убедиться в формальной правильности программы. Кроме синтаксиса, заданного КС-грамматикой, должны соблюдаться дополнительные правила: об обязательных описаниях объектов программы, соответствии типов и т. п. Выполнение этих правил проверяется с помощью таблиц, заполняемых в ходе трансляции, и составляет суть контекстного анализа. Синтаксический анализатор формирует промежуточное представление программы в форме обратной польской записи, семантического дерева, четверок.

Оптимизатор. В простых компиляторах может отсутствовать. Суть оптимизации состоит в эквивалентном преобразовании программы (промежуточного представления) с целью улучшения ее эффективности: увеличения быстродействия и уменьшения расхода памяти. Оптимизация, выполняемая на уровне промежуточного представления, не зависит от выходного языка компилятора, является машинно-независимой.

Генератор кода. Отвечает за формирование результирующей программы — машинных команд. В составе генератора кода также могут быть части, выполняющие машинно-зависимую оптимизацию. Программу, получаемую в результате компиляции, обычно называют объектной программой. Это может быть не полностью готовая к выполнению машинная программа, а отдельный модуль, который еще должен быть скомпонован с другими.

Таблицы компилятора. Компилятор содержит множество таблиц, с которыми взаимодействуют его модули: таблица зарезервированных слов, таблица имен, таблица типов и др. На поиск в таблицах тратится значительная часть времени работы транслятора.

Блоки компилятора, зависящие от входного языка, выполняющие анализ исходной программы и ее преобразование в промежуточное

представление, образуют его анализирующую, входную, фронтальную часть (по-английски front-end). Блоки, ответственные за формирование выходной программы и не зависящие от входного языка, образуют синтезирующую, выходную часть (back-end)[63].

Многопроходные и однопроходные трансляторы

Показанные на рисунке 3.2 блоки транслятора (кроме драйвера исходного текста) можно рассматривать как последовательно выполняемые модули, осуществляющие отдельные фазы трансляции. Вначале лексический анализатор, прочитывая исходную программу от начала до конца, формирует последовательность лексем — лексическую свертку и сохраняет ее. Синтаксический анализатор вступает в работу по окончании работы сканера. Далее последовательно запускаются оптимизатор и генератор кода. Компилятор, работающий по такой схеме, называется *многопроходным*. Проходом считается прочтение транслятором программы от начала до конца в ее исходной или одной из промежуточных форм: лексической свертки, промежуточных представлений. Не обязательно разбиение на проходы должно соответствовать разделению на блоки, показанные на рисунке 3.2. Некоторые из блоков и фаз трансляции могут объединяться в один проход и наоборот, отдельные фазы могут выполняться в несколько проходов.

Последовательная работа основных блоков компилятора совсем не обязательна. Эти блоки можно рассматривать как логические части, действующие попеременно и выполняющие трансляцию за один просмотр программы. Ведущую роль при этом выполняет синтаксический анализатор. Обращаясь за лексемами к сканеру, он выполняет анализ и, распознав определенную часть программы, вызывает генератор кода для формирования порции машинных команд.

Несколько измененная схема, отражающая такой способ взаимодействия частей в однопроходном трансляторе, показана на рисунке 3.3. В таблице 3.1 представлены преимущества и недостатки однопроходной и многопроходной организации.

---

[63] К сожалению, устоявшиеся русские термины, эквивалентные полезным английским «front-end» и «back-end» не сформировались и, наверное, уже не сформируются. В устных беседах «компиляторостроители» обычно так и говорят: «фронт-энд», «бэк-энд». Ввести такие кальки с английского в письменную речь я не рискну.

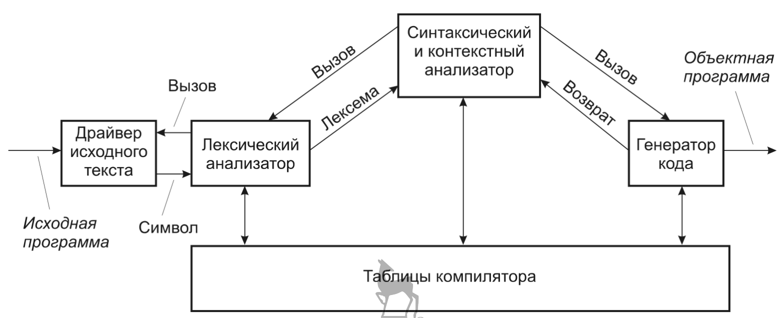


Рис. 3.3. Структура однопроходного транслятора

Таблица 3.1. Сравнение однопроходных и многопроходных трансляторов

	Однопроходный	**Многопроходный**
Преимущества	Простота. Быстрая работа. Отсутствие необходимости в промежуточных представлениях	Нетребовательность к памяти. Лучшее разделение на модули. Лучшие возможности для оптимизации программы. Независимость анализатора от выходного языка. Независимость генератора от входного языка
Недостатки	Затруднена оптимизация. Плохое разделение анализатора и генератора кода. Дополнительные требования к входному языку	Сложность. Более медленная работа. Необходимость промежуточных представлений

Первые компиляторы с языков высокого уровня были многопроходными. Это объяснялось тем, что ресурсы имевшихся компьютеров (объем оперативной памяти в первую очередь) просто не позволяли выполнить всю работу за один раз. Сам компилятор не помещался

целиком в память, и его разбивали на части, которые работали последовательно. Кроме того, однопроходная схема предполагает, что в оперативной памяти в ходе трансляции должна удерживаться значительная часть формируемого объектного кода (как минимум, код одной процедуры).

Количество частей и проходов могло быть довольно большим. Так, один из первых отечественных трансляторов с Алгола-60 ТА-2М состоял из 16 блоков и выполнял 16 проходов. В современных условиях уже нет технических ограничений, препятствующих однопроходной трансляции.

Не всякий язык программирования приспособлен для того, чтобы любая написанная на нем программа могла быть откомпилирована за один проход. Для однопроходной компиляции в частности требуется, чтобы употреблению объекта программы предшествовало его описание.

Компилятор языка «О»

Учебный компилятор языка «О» будет однопроходным. Простота устройства нам сейчас важнее возможностей оптимизации и скорости результирующей программы. Цель — написать работающий компилятор. И быть при этом в возможно большей степени уверенными, что работает он правильно[64]. Программировать будем на Паскале, который в этой книге играет роль основного языка.

Приступим. Вот первый эскиз главной программы (листинг 3.1).

Листинг 3.1. Компилятор. Первый шаг детализации

```
program O;
{Учебный компилятор языка O}
   …
begin
   WriteLn('Компилятор языка O');
   Init;      {Инициализация}
   Compile;   {Компиляция    }
   Done;      {Завершение    }
end.
```

[64] Среди всех мыслимых качеств программы главное, безусловно, — ее работоспособность. Как справедливо замечено, если не требуется, чтобы программа правильно работала, ее можно сделать сколь угодно быстрой и компактной. И, добавлю, генерирующей сколь угодно быстрый и компактный код.

Может показаться, что этот шаг банален. Это не так. Разбив задачу на три последовательно выполняемые части, мы можем поочередно сконцентрироваться на каждой из них, забыв на время про остальные.

Подумаем об инициализации. Она может начинаться с подготовки текста исходной программы к чтению компилятором. Эту работу в предшествующих примерах выполняла процедура `ResetText`. Она же послужит нам и в этот раз. Процедура `ResetText` будет частью драйвера исходного текста.

По окончании компиляции возможно потребуется выполнить действия, завершающие работу с источником исходного текста: закрыть файл, отправить сообщение окну текстового редактора. За это будет отвечать процедура `CloseText`, также относящаяся к драйверу исходного текста.

Перед тем, как записать уточненную версию главной программы учтем, что `Compile` — основная процедура компилятора — это часть модуля синтаксического анализатора, который при однопроходной трансляции играет ведущую роль, координируя работу остальных частей. В нашей программе (листинг 3.2) синтаксический анализатор (parser) будет оформлен как отдельный модуль с названием `OPars`. Драйвер исходного текста назовем `OText`.

Листинг 3.2. Компилятор. Первый компилируемый вариант

```
program O;
{Учебный компилятор языка О}
uses
   OText, OPars;
procedure Init;
begin
   ResetText;
end;

procedure Done;
begin
   CloseText;
end;

begin
   WriteLn('Компилятор языка О');
   Init;      {Инициализация}
   Compile;   {Компиляция    }
   Done;      {Завершение    }
end.
```

В отличие от первого эскиза, это уже вариант, который может быть откомпилирован при условии, что имеются модули OText и OPars.

Для начала в роли процедуры Compile модуля OPars используем заглушку (листинг 3.3).

Листинг 3.3. Модуль синтаксического анализатора с заглушкой

```
unit OPars;
{Распознаватель}
interface
procedure Compile;

{=====================================================}

implementation
procedure Compile;
begin
   {Тут пусто. Это заглушка}
end;
end.
```

Вспомогательные модули компилятора

Я не буду программировать здесь полностью драйвер исходного текста, как и некоторые другие вспомогательные модули. Определим только их программные интерфейсы.

Драйвер исходного текста

Драйвер исходного текста (модуль OText, листинг 3.4) непосредственно взаимодействует с транслируемой программой. Другие части компилятора (в первую очередь сканер) обращаются к драйверу за очередным символом. Чтение символа выполняет процедура NextCh. Прочитанный символ она помещает в глобальную переменную Ch, экспортируемую драйвером. В программный интерфейс модуля OText входят также уже использовавшиеся процедуры ResetText и CloseText.

Листинг 3.4. Интерфейс драйвера исходного текста

```
unit OText;
{Драйвер исходного текста}

interface

const
```

```
   chSpace = ' ';      {Пробел        }
   chTab   = chr(9);   {Табуляция     }
   chEOL   = chr(10);  {Конец строки}
   chEOT   = chr(0);   {Конец текста}

var
   Ch     : char;     {Очередной символ        }
   Line   : integer;  {Номер строки            }
   Pos    : integer;  {Номер символа в строке}

procedure ResetText;
procedure CloseText;
procedure NextCh;
{=====================================================}
```

Определены константы chSpace, chTab и chEOL[65], обозначающие пробел, табуляцию и конец строки, а также признак конца текста chEOT.

Драйвер исходного текста ведет подсчет строк и символов, помещая номер текущей строки в переменную Line, а номер текущего символа в строке — в переменную Pos. Эти значения могут использоваться при выдаче сообщений об ошибках.

После выполнения ResetText текущим (содержащимся в переменной Ch) должен стать первый символ компилируемой программы. Это может быть и незначащий символ — пробел, табуляция, разрыв строки. По ходу чтения исходный текст может печататься.

Вот пример запуска нашего компилятора:

```
>O Primes.o
Компилятор языка О
>
```

Не полностью готового, а состоящего только из главной программы (листинг 3.2), модуля OPars с заглушкой вместо Compile (листинг 3.3) и драйвера исходного текста. А это пример неправильного вызова:

[65] То, что конец строки представлен значением chEOL, совсем не означает, что, строки реального текста обязательно разделяются одним специальным символом равным chr(10). Ситуация лишь представляется такой другим частям компилятора. Драйвер исходного текста, обнаружив границу строк ему одному известным способом, помещает в переменную Ch значение chEOL.

```
>O.exe
Компилятор языка О
Формат вызова:
   О <входной файл>
```

Обработка ошибок

За нее отвечает модуль `OError` (листинг 3.5). Он предоставляет другим частям транслятора процедуру `Error`, которая печатает переданное ей как параметр сообщение об ошибке, указывая место ошибки в исходном тексте. После этого процедура `Error` вызовом `Halt` прекращает работу компилятора. Такая простая реакция на ошибки нас вполне устраивает, поскольку создаваемый компилятор не является частью какой-либо интегрированной системы.

Листинг 3.5. Интерфейс модуля обработки ошибок

```
unit OError;
{Обработка ошибок}

interface

procedure Error(Mes: string);
procedure Expected(Mes: string);
procedure Warning(Mes: string);

{=====================================================}
```

Поскольку большинство сообщений компилятора об ошибках начинаются словом «Ожидается», предусмотрена процедура `Expected`, которая делает то же, что и `Error`, но сама печатает слово «Ожидается», избавляя нас от необходимости записывать его каждый раз. Потребуется также возможность печатать предупреждения. Для этого предусмотрена процедура `Warning`. В отличие от `Error` и `Expected`, она не останавливает работу программы.

Лексический анализатор (сканер)

Лексический анализатор решает следующие задачи:

- Выполняет «сборку» лексем из отдельных символов входного текста. Лексемы — это неделимые единицы программы — имена, числа, зарезервированные слова, разделители и знаки операций, в том числе состоящие из нескольких символов, как, например, «`:=`».
- Удаляет комментарии и пробельные символы (пробел, табуляция, конец строки).

Рассмотрим для примера такой фрагмент программы на языке «О» (Оберон):

```
i := 2;
WHILE i <= n DO
```

В нем содержится 9 лексем: три лексемы «имя», лексемы «зарезервированное слово **WHILE**» и «слово **DO**», а также лексемы «присваивание», «меньше или равно», «целое число», «точка с запятой».

При использовании в компиляторе отдельного блока сканера грамматика языка программирования становится двухуровневой. С точки зрения сканера программа — это просто последовательность лексем. Правила записи лексем, равно как и синтаксис последовательности лексем могут быть заданы автоматной грамматикой. Терминалами этой грамматики являются отдельные знаки, в том числе пробельные символы. Это первый уровень определения языка — *лексическая грамматика*. Второй уровень — *синтаксическая грамматика*, которая используется синтаксическим анализатором. Терминалами синтаксической грамматики являются лексемы.

Использование сканера обусловлено такими причинами:

- Упрощение синтаксического анализатора. Синтаксический анализатор имеет дело не с отдельными символами, а с более крупными и содержательными единицами, избавляется от необходимости обрабатывать пробельные символы и комментарии.
- Повышается эффективность следующих за сканером фаз трансляции. Элементы программы на входе синтаксического анализатора представляются не строками переменной длины, а данными фиксированного размера. Размер лексической свертки может быть меньше размера исходной программы.
- Естественным образом решается вопрос о пробельных символах. Пробелы, табуляции и концы строк запрещаются внутри лексем (кроме пробелов и табуляций в символьных строках[66]) и разрешаются между ними.
- Обеспечивается независимость следующих за сканером фаз трансляции от конкретного представления исходной программы и используемого набора символов. Облегчается изменение лексики язы-

[66] В некоторых языках, например в Си#, внутри символьных литералов разрешается помещать и разрыв строки, в некоторых языках не допускаются табуляции. В языке «О» символьные строки вообще не предусмотрены.

ка. Замена или перевод на другой язык служебных слов не требуют изменения синтаксического анализатора и последующих блоков.

Виды и значения лексем

Взаимодействие сканера с последующими частями транслятора в многопроходном и однопроходном трансляторе строится несколько по-разному.

В многопроходном трансляторе лексический анализатор должен целиком сформировать кодированную последовательность лексем — лексическую свертку[67]. Для многих лексем достаточно было бы указать в свертке только их вид, то есть сообщить, что это за лексема. Виды лексем могут кодироваться целыми числами. Например, для лексем «присваивание» или «слово **WHILE**», достаточно указать, что это именно они, записав в свертку соответствующий код. Но для таких лексем как «число» и «имя» указания их вида не достаточно. Синтаксическому анализатору безразлично, какое конкретно число записано в программе, но к моменту генерации кода эта информация потребуется. В связи с этим лексеме, наряду с ее видом, приписывается значение, а лексическая свертка представляется последовательностью пар вид-значение.

Значением лексемы «число» может быть величина этого числа или ссылка на запись в таблице констант. Значение лексемы «имя» — строка, содержащая символы конкретного имени, или ссылка на запись в таблице имен. Использование ссылок вместо самих значений позволяет обойтись в лексической свертке полями фиксированного размера.

В однопроходном трансляторе все проще. Поскольку формировать лексическую свертку целиком не требуется, сведения о текущей лексеме могут передаваться синтаксическому анализатору с помощью единственной переменной. Для лексем, которым приписывается значение, можно предусмотреть одно или несколько полей в записи о лексеме (или несколько отдельных переменных) — по одному для значений разных типов. Одно поле (переменная) хранит значения целых чисел, другое — вещественных и т. д. Вместо самих значений

[67] Точнее, не в любом многопроходном, а лишь в том, где лексический анализ выполняется отдельным проходом, что в современных условиях вряд ли актуально.

могут использоваться ссылки на записи в соответствующих таблицах, которые в этом случае должен формировать сканер.

Лексический анализатор языка «О»

В нашем компиляторе, который мы пишем на Паскале, лексический анализатор будет реализован в виде отдельного модуля `OScan`.

Виды лексем языка «О»

Лучшим вариантом кодирования видов лексем при программировании на Паскале будет использование перечислимого типа.

```
type
   tLex = (lexNone, …
```

Виды лексем будут константами типа `tLex`. Первая предусмотренная нами константа — первый вид лексем — `lexNone` — «никакая» лексема. Это особое значение будет соответствовать тем служебным словам Оберона-2, которые не используются в языке «О», например, **RECORD**, **IS**. Лексема вида `lexNone` не может присутствовать ни в одной правильной программе на языке «О». Продолжаем:

```
   tLex =  (lexNone, lexName, lexNum,
            lexMODULE, lexIMPORT, lexBEGIN, lexEND,
            lexCONST, lexVAR, lexWHILE, lexDO,
            lexIF, lexTHEN, lexELSIF, lexELSE,
   …
```

`lexName` и `lexNum` — это лексемы «имя» и «число». Сканер должен формировать для этих лексем еще и значения.

Служебным словам **MODULE**, **IMPORT**, **BEGIN** и другим соответствуют одноименные виды лексем. Значения для этих лексем не нужны. Если сканер выдает `lexBEGIN`, никакой другой информации от него не требуется.

Далее в определении типа `tLex` будут следовать лексемы, соответствующие знакам операций. Здесь возможны различные решения. Хотя знаки плюс и минус имеют разный смысл, с точки зрения синтаксического анализатора они совершенно равноправны и всегда обрабатываются им одинаково, как равноправны знаки операций типа умножения: «*», **DIV**, **MOD**, и отношения «=», «#», «<», «<=», «>», «>=». Поэтому можно отнести знаки операций типа сложения к одному виду, типа умножения — к другому, знаки отношений — к третьему. При этом соответствующие лексемы надо будет снабжать зна-

чениями, позволяющими на стадии генерации кода отличить плюс от минуса и умножение от деления.

Но можно каждый из перечисленных знаков относить к самостоятельному виду лексем.

При использовании пары вид-значение работа синтаксического анализатора немного упростится. Для проверки того, что текущая лексема это, например, операция типа сложения, придется сделать только одно сравнение на равенство. Если каждый знак закодирован по-особому, потребуется проверка на принадлежность текущей лексемы множеству значений.

Примем простое, «лобовое» решение — отнесем каждый знак к особому виду.

```
...
lexMult, lexDIV, lexMOD, lexPlus, lexMinus,
lexEQ, lexNE, lexLT, lexLE, lexGT, lexGE,
...
```

Обратите внимание на имена констант. Лексема, обозначающая «–», названа `lexMinus` — «минус», а не, скажем, `lexSub` — «вычитание», поскольку знак «–» может обозначать разные операции: одноместный минус — это перемена знака, двуместный — вычитание. Названия лексем, обозначающих отношения, выбраны с учетом традиции, восходящей к Фортрану: EQ (equal) — равно; NE (not equal) — не равно; LT (less than) — меньше чем; LE (less than or equal) — меньше или равно; GT (greater than) — больше чем; GE (greater than or equal) — больше или равно.

Наконец, замыкают определение типа `tLex` константы, обозначающие лексемы-разделители:

```
...
lexDot, lexComma, lexColon, lexSemi, lexAss,
lexLPar, lexRPar,
lexEOT);
```

Dot — точка; comma — запятая; colon — двоеточие; semi(colon) — точка с запятой; ass(ignment) — присваивание; l(eft) par(enthesis) — левая круглая скобка; r(ight) par(enthesis) — правая круглая скобка. Константа `lexEOT`, означает лексему «конец текста» (её не следует путать с символом `chEOT`).

Программирование сканера

Правила записи лексем могут быть заданы автоматной грамматикой. Поэтому на роль сканера подошел бы конечный автомат. Мы могли бы изобразить диаграмму его переходов, построить таблицу. Другой вариант — программировать лексический анализатор по синтаксическим диаграммам. Но и в построении диаграмм нет реальной необходимости. Правила записи лексем языка «О» настолько просты, что сканер можно программировать «просто так». Построить диаграмму потребуется лишь для комментария. Кстати сказать, комментарии в языке «О» могут быть вложенными, поэтому их синтаксис задается контекстно-свободной, но не автоматной грамматикой.

В компиляторе, где лексический анализ не выделен в отдельный проход, собственно сканер — это процедура, выдающая по запросу синтаксического анализатора очередную лексему. Назовем эту процедуру `NextLex`. Задача `NextLex` состоит в том, чтобы, считывая символы, определить вид очередной лексемы и ее значение, (если для лексем данного вида значение требуется).

Перед первым вызовом процедуры `NextLex` текущим является первый символ входного текста. Он может быть и пробельным. Сканер должен пропустить пробелы, предшествующие лексеме, если они есть, прочитать символы лексемы и оставить текущим символ, следующий за лексемой. Такие же действия выполняются при каждом следующем вызове. Если запись лексемы содержит ошибку, сканер выдает соответствующее сообщение.

Начало процедуры `NextLex` будет таким:

```
procedure NextLex;
begin
   while Ch in [chSpace, chTab, chEOL] do NextCh;
   LexPos := Pos;
```

Напомню, что переменные `Ch` и `Pos`, процедура `NextCh` и константы `chSpace`, `chTab` и `chEOL` импортированы сканером из модуля `OText` (см. листинг 3.4).

Цикл **`while`** выполняет пропуск пробельных символов. По выходе из цикла текущим будет первый знак очередной лексемы (если какая-то лексема может начинаться с этого знака). Номер этого символа в строке хранится в переменной `Pos`. Полезно запомнить этот номер. Для этой цели служит переменная сканера `LexPos`. Она экспортиру-

ется сканером и может использоваться в модуле `OError`, для того, чтобы иметь возможность в диагностическом сообщении указать на начало лексемы, вызвавшей ошибку. Если для этих целей употребить `Pos`, то указать можно будет лишь на символ, следующий за лексемой, или на последний символ лексемы.

Далее, в зависимости от значения текущего символа (который предположительно является первым символом очередной лексемы), решаем, к какому виду эта лексема должна быть отнесена и при необходимости определяем её значение. Первый вариант выбора относится к случаю, когда символ — это буква, второй — когда символ — цифра.

```
case Ch of
'A'..'Z', 'a'..'z':
   Ident;
'0'..'9':
   Number;
…
```

С буквы может начинаться лексема «имя» (`lexName`) или одно из ключевых слов. Чтение символов лексемы, следующих за первой буквой, и определение того, является ли лексема именем или зарезервированным словом отложим, поручив процедуре `Ident`. Различать имена и ключевые слова, процедура `Ident` будет с помощью таблицы служебных слов.

С цифры в языке «О» (и многих других языках) может начинаться только число[68]. Чтение его последующих цифр и вычисление значения выполнит процедура `Number`.

Вид текущей лексемы (имя, служебное слово, число и т. д.) будем записывать в глобальную переменную `Lex`, которая экспортируется модулем `OScan`. Для значений числовых литералов предусмотрим глобальную переменную `Num`, а символы, составляющие имя, процедура `Ident` будет помещать в глобальную переменную `Name`.

---

[68] Здесь полезно будет ввести в обиход понятие «литерал». В данном случае — это число, записанное с помощью цифр. Это числовой литерал. В то же время в программе могут быть числовые константы, которым даны символические имена, и которые литералами не являются. Литерал — это константа, записанная как есть, буквально. Наряду с числовыми, могут быть строковые, символьные и другие литералы.

Перед тем как продолжить распознавание лексем, зафиксируем интерфейс модуля `OScan` (листинг 3.6), который уже вполне определился.

Листинг 3.6. Интерфейс модуля сканера

```
unit OScan;
{ Сканер }
interface

const
   NameLen = 31; {Наибольшая длина имени}

type
   tName = string[NameLen];
   tLex  = (lexNone, lexName, lexNum,
         lexMODULE, lexIMPORT, lexBEGIN, lexEND,
         lexCONST, lexVAR, lexWHILE, lexDO,
         lexIF, lexTHEN, lexELSIF, lexELSE,
         lexMult, lexDIV, lexMOD, lexPlus, lexMinus,
         lexEQ, lexNE, lexLT, lexLE, lexGT, lexGE,
         lexDot, lexComma, lexColon, lexSemi, lexAss,
         lexLpar, lexRpar,
         lexEOT);

var
   Lex    : tLex;     {Текущая лексема              }
   Name   : tName;    {Строковое значение имени     }
   Num    : integer;  {Значение числовых литералов}
   LexPos: integer;   {Позиция начала лексемы       }

procedure InitScan;
procedure NextLex;
{=====================================================}
```

Несколько пояснений. Определив тип `tName` для имён и установив максимальную длину имени, мы ввели ограничение, которого не было в спецификации языка. Это особенность нашей реализации языка «О». Такие ограничения представляются вполне допустимыми. Другим решением могла быть попытка не вводить предельной длины имени. Это, несомненно, привело бы к усложнению способа хранения имен, но вряд ли принесло бы реальную пользу, поскольку принятое ограничение в 31 символ достаточно для подавляющего большинства программ. Наш компилятор будет предупреждать о превышении этого лимита.

Процедура `InitScan`, упомянутая в интерфейсе модуля, будет отвечать за приведение сканера в исходное состояние. В числе ее задач — прочитать первую лексему программы и заполнить таблицу служебных слов.

Продолжим распознавание лексем. При выборе порядка следования вариантов в **case** можно учитывать частоту использования лексем разных видов в программах. Если лексемы, встречающиеся чаще, распознавать в первую очередь, сканер, возможно, будет работать несколько быстрей[69]. Одна из самых употребляемых лексем — точка с запятой, часто используются точка и запятая:

```
';':
   begin
      NextCh;
      Lex := lexSemi;
   end;
'.':
   begin
      NextCh;
      Lex := lexDot;
   end;
',':
   begin
      NextCh;
      Lex := lexComma;
   end;
```

Распознавание этих лексем элементарно. Переменная `Lex` просто получает соответствующее значение. Вызов `NextCh` делает текущим следующий за лексемой символ.

Немного сложнее обстоит дело с обработкой двоеточия. С этого знака могут начинаться лексемы двух видов: собственно двоеточие (лексема `lexColon`) и знак присваивания (`lexAss`). Чтобы различить эти два случая достаточно прочитать еще один символ. Если это будет знак «=», значит лексема — присваивание, иначе — двоеточие. В первом случае перед выходом из `NextLex` надо прочитать еще один символ.

---

[69] Оптимизирующие компиляторы создают код для оператора **case** (и подобных ему в других языках), основанный на использовании таблицы переходов. В этом случае скорость выполнения **case** не зависит от упорядочения вариантов.

```
':':
   begin
      NextCh;
      if Ch = '=' then begin
         NextCh;
         Lex := lexAss;
         end
      else
         Lex := lexColon;
   end;
```

Следующие лексемы — знаки операций отношения. Их шесть видов. Но начинаться эти лексемы могут только с четырех различных символов:

```
'=':
   begin
      NextCh;
      Lex := lexEQ;
   end;
'#':
   begin
      NextCh;
      Lex := lexNE;
   end;
'<':
   begin
      NextCh;
      if Ch='=' then begin
         NextCh;
         Lex := lexLE;
         end
      else
         Lex := lexLT;
   end;
'>':
   begin
      NextCh;
      if Ch='=' then begin
         NextCh;
         Lex := lexGE;
         end
      else
         Lex := lexGT;
   end;
```

Открывающая круглая скобка может представлять саму себя (лексема `lexLPar`), но с этого же знака начинается комментарий. Различим эти два случая, прочитав еще один символ:

```
'(':
   begin
      NextCh;
      if Ch = '*' then begin
         Comment;
         NextLex; {Рекурсия}
         end
      else
         Lex := lexLpar;
   end;
```

Обработку комментария (которая состоит в пропуске всех его символов) выполнит процедура `Comment`, которую запишем позже. А пока отметим, что в момент начала её работы текущим является символ «*». После пропуска комментария процедура `NextLex` вызывается рекурсивно — сам комментарий лексемой не является, а `NextLex` не может завершить работу, не прочитав лексемы.

Распознавание остальных видов лексем не составляет труда:

```
')':
   begin
      NextCh;
      Lex := lexRpar;
   end;
'+':
   begin
      NextCh;
      Lex := lexPlus;
   end;
'-':
   begin
      NextCh;
      Lex := lexMinus;
   end;
'*':
   begin
      NextCh;
      Lex := lexMult;
   end;
chEOT:
   Lex := lexEOT;
```

```
    else
       Error('Недопустимый символ');
    end {case};
 end {NextLex};
```

Если символ, ставший текущим после пропуска пробелов, не совпадает ни с одним из перечисленных вариантов, сканер сообщает о лексической ошибке.

Распознавание имен

Выполняется процедурой `Ident` (листинг 3.7). В ее задачу входит считывание символов имени и их запись в переменную сканера `Name`. Если считанное имя не совпадает ни с одним из зарезервированных слов, процедура `Ident` должна занести значение `lexName` в переменную сканера `Lex`, в противном случае `Lex` получает значение, соответствующее зарезервированному слову. В начале работы процедуры `Ident` переменная `Ch` уже содержит первую букву идентификатора.

Листинг 3.7. Сканирование идентификаторов

```
procedure Ident;
var
   i : integer;
begin
   Name := '';
   i := 0;
   repeat
      if i < NameLen then begin
         i := i + 1;
         Name := Name + Ch;
         end
      else
         Error('Слишком длинное имя');
      NextCh;
   until not (Ch in ['A'..'Z', 'a'..'z', '0'..'9']);
   Lex := TestKW;      {Проверка на ключевое слово}
end;
```

При превышении допустимой длины имени сканер сообщает об ошибке и прекращает работу компилятора. Другим возможным решением было бы продолжение чтения символов имени, без их записи в переменную `Name`. В этом случае компилятор будет разрешать использовать имена произвольной длины, но различаться они будут по

первым NameLen символам[70]. Первый, более строгий вариант представляется предпочтительным, поскольку исключает какие-либо недоразумения.

Таблица ключевых слов

Определение вида лексемы по содержимому переменной Name выполняет функция TestKW (testing keywords — проверка ключевых слов), значением которой является вид лексемы. Функция TestKW использует таблицу, ключ поиска в которой — строка, содержащая идентификатор. Для каждого служебного слова в таблице записан вид соответствующей лексемы (значение типа tLex). Если для какого-то имени поиск в таблице оканчивается неудачей (имя не найдено, поскольку не совпадает ни с одним из ключевых слов), функция TestKW возвращает значение lexName.

Таблица ключевых слов может быть организована по-разному. При ее конструировании можно стремиться сделать поиск по возможности быстрым. Обращение к таблице служебных слов происходит всякий раз, когда в компилируемой программе встречается идентификатор, поэтому от времени поиска заметно зависит скорость работы всего компилятора. Подходящими вариантами для таблицы будут двоичный поиск в упорядоченном по ключам массиве, перемешанная (хэш) таблица, таблица, организованная в виде двоичного дерева. Можно учесть, что содержание таблицы неизменно — в ней содержатся 34 зарезервированных слова языка «О» (Оберона-2). Это позволяет заранее разместить их так, чтобы максимально ускорить поиск. Разумно принять во внимание и частоту использования различных служебных слов в реальных программах.

В начале работы компилятора таблица зарезервированных слов заполняется данными. Занесение отдельного слова в таблицу будем выполнять с помощью процедуры EnterKW. При ее вызове указываются ключевое слово и вид лексемы. Вот так может выглядеть занесение в таблицу первых трех слов:

```
EnterKW('ARRAY', lexNone);
EnterKW('BY',    lexNone);
EnterKW('BEGIN', lexBEGIN);
```

[70] Такой подход используется в диалектах языка Паскаль компании Borland (Inprise, Embarcadero), где значащими являются первые 63 или 255 символов имени.

Устройство процедуры `EnterKW` и порядок занесения слов зависят от выбранного способа организации таблицы.

ЗАДАНИЕ

> Сконструируйте и реализуйте таблицу служебных слов так, чтобы скорость работы сканера была по возможности наибольшей. При этом желательно не жертвовать наглядностью программы и не превышать разумного расхода памяти.

Здесь же мы используем простейший вариант организации таблицы служебных слов — массив, в котором выполняется последовательный, линейный поиск. Это самый медленный способ. Но его простота позволяет быстрее продвинуться в разработке компилятора, сохраняя уверенность в правильной работе сканера. В дальнейшем линейный поиск можно заменить более эффективным алгоритмом.

Предусмотрим в секции реализации модуля `OScan` следующие описания:

```
const
   KWNum = 34;    {Размер таблицы}
type
   tKeyWord = string[9]; {Длина слова PROCEDURE}
var
   nkw: integer; {Число занесенных в таблицу слов}
   KWTable: array [1..KWNum] of
      record
         Word: tKeyWord;
         Lex : tLex;
      end;
```

Заполнение таблицы выполняется при инициализации сканера процедурой `InitScan` (листинг 3.8).

Листинг 3.8. Инициализация сканера

```
procedure InitScan;
begin
   nkw := 0; {Вначале таблица пуста}

   EnterKW('ARRAY',     lexNone);
   EnterKW('BY',        lexNone);
   EnterKW('BEGIN',     lexBEGIN);
   EnterKW('CASE',      lexNone);
   EnterKW('CONST',     lexCONST);
```

```
   EnterKW('DIV',       lexDIV);
   EnterKW('DO',        lexDO);
   EnterKW('ELSE',      lexELSE);
   EnterKW('ELSIF',     lexELSIF);
   EnterKW('END',       lexEND);
   EnterKW('EXIT',      lexNone);
   EnterKW('FOR',       lexNone);
   EnterKW('IF',        lexIF);
   EnterKW('IMPORT',    lexIMPORT);
   EnterKW('IN',        lexNone);
   EnterKW('IS',        lexNone);
   EnterKW('LOOP',      lexNone);
   EnterKW('MOD',       lexMOD);
   EnterKW('MODULE',    lexMODULE);
   EnterKW('NIL',       lexNone);
   EnterKW('OF',        lexNone);
   EnterKW('OR',        lexNone);
   EnterKW('POINTER',   lexNone);
   EnterKW('PROCEDURE', lexNone);
   EnterKW('RECORD',    lexNone);
   EnterKW('REPEAT',    lexNone);
   EnterKW('RETURN',    lexNone);
   EnterKW('THEN',      lexTHEN);
   EnterKW('TO',        lexNone);
   EnterKW('TYPE',      lexNone);
   EnterKW('UNTIL',     lexNone);
   EnterKW('VAR',       lexVAR);
   EnterKW('WHILE',     lexWHILE);
   EnterKW('WITH',      lexNone);

   NextLex; {Чтение первой лексемы}
   {возможно только после заполнения таблицы}
end;
```

Вызов InitScan следует предусмотреть при инициализации компилятора, за которую отвечает процедура Init в основной программе:

```
procedure Init;
begin
   ResetText;
   InitScan;
end;
```

Занесение в таблицу данных об одном служебном слове выполняет процедура EnterKW:

```
procedure EnterKW(Name: tKeyWord; Lex: tLex);
begin
   nkw := nkw + 1;
   KWTable[nkw].Word := Name;
   KWTable[nkw].Lex := Lex;
end;
```

Функция `TestKW` (листинг 3.9) выполняет линейный поиск значения `Name` в таблице `KWTable`:

Листинг 3.9. Линейный поиск в таблице ключевых слов

```
function TestKW: tLex;
var
   i : integer;
begin
   i := nkw;
   while (i>0) and (Name<>KWTable[i].Word) do
      i := i-1;
   if i>0 then
      TestKW := KWTable[i].Lex
   else
      TestKW := lexName;
end;
```

Сканирование числовых литералов

В языке «О» предусмотрены только целые числа. Их обработка выполняется достаточно просто (листинг 3.10). Вычисленное значение числового литерала записывается в переменную сканера `Num`.

Листинг 3.10. Сканирование чисел

```
procedure Number;
var
   d : integer;
begin
   Lex := lexNum;
   Num := 0;
   repeat
      d := ord(Ch) - ord('0');
      if (Maxint - d) div 10 >= Num then
         Num := 10*Num + d
      else
         Error('Слишком большое число');
      NextCh;
   until not (Ch in ['0'..'9']);
end;
```

При вызове процедуры Number переменная Ch уже содержит первую цифру числа.

Пропуск комментариев

Выполняется процедурой Comment. Ее задача — прочитать все символы, включая закрывающую комментарий пару «*)». В языке «О», как и в Обероне, комментарии могут быть вложенными, поэтому должно быть правильно учтено соответствие открывающих «(*» и закрывающих «*)» пар символов.

При обработке комментария необходимо учитывать, что текст программы может закончиться до того, как встретятся символы «*)». Поэтому нужна явная проверка символа «конец текста». Синтаксическая диаграмма для комментария представлена на рисунке 3.4.

Вызов процедуры Comment (листинг 3.11) происходит, когда текущим символом является «*», поэтому формально удобно считать, что комментарий — это конструкция, начинающаяся символом «*». Такому соглашению подчиняется и рекурсивное обращение к комментарию, которое можно видеть на диаграмме.

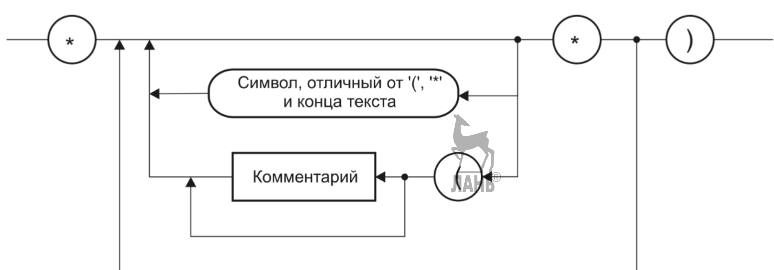


Рис. 3.4. Синтаксическая диаграмма комментария

В соответствии со структурой диаграммы распознающая процедура содержит два вложенных цикла и рекурсивно обращается к себе.

Листинг 3.11. Рекурсивный пропуск комментариев

```
procedure Comment;
begin
   NextCh;
   repeat
      while (Ch <> '*') and (Ch <> chEOT) do
```

```
         if Ch = '(' then begin
            NextCh;
            if Ch = '*' then Comment;
            end
         else
            NextCh;
      if Ch = '*' then NextCh;
   until Ch in [')', chEOT];
   if Ch = ')' then
      NextCh
   else begin
      LexPos := Pos;
      Error('Не закончен комментарий');
   end;
end;
```

Чтобы при выдаче диагностического сообщения «Не закончен комментарий» было правильно указано место ошибки (место, где закончился текст), переменной `LexPos` присваивается значение `Pos`, равное номеру текущего символа в строке. В обычном случае `LexPos` обозначает номер символа, начинающего лексему, но комментарий — не лексема, может занимать несколько строк, поэтому сохранять значение `LexPos`, соответствующее месту, где комментарий начался, было бы неправильно.

Пропуск комментариев может быть выполнен и по-другому. Программа, приведенная в листинге 3.12, использует счетчик уровня вложенности комментариев `Level`. Счетчик увеличивается на единицу при входе в комментарий и уменьшается при выходе из него. Нулевое значение `Level` соответствует положению вне комментария.

Листинг 3.12. Нерекурсивный пропуск комментариев

```
procedure Comment;
var
   Level : integer;
begin
   Level := 1;
   NextCh;
   repeat
      if Ch = '*' then begin
         NextCh;
         if Ch = ')' then begin
            Level := Level - 1;
            NextCh
```

```
         end;
         end
      else if Ch = '(' then begin
         NextCh;
         if Ch = '*' then begin
            Level := Level + 1;
            NextCh
         end;
         end
      else {if Ch <> chEOT then}
         NextCh;
   until (Level = 0) or (Ch = chEOT);
   if Level <> 0 then begin
      LexPos := Pos;
      Error('Не закончен комментарий');
   end;
end;
```

Работа этой процедуры подобна поведению автомата с магазинной памятью. Роль стека здесь исполняет счетчик `Level`. В стек как бы помещаются открывающие скобки комментария, а каждая закрывающая скобка удаляет из стека соответствующую открывающую. Но, поскольку в стек всегда заносятся одни и те же элементы, запоминать их самих нет нужды, достаточно подсчитывать их количество.

Тестирование сканера

Лексический анализатор не выполняет в нашем компиляторе самостоятельного просмотра исходной программы. Сканер — лишь процедура, которую вызывает синтаксический анализатор, если ему требуется очередная лексема. Чтобы не откладывать тестирование сканера до момента, когда будет готов синтаксический анализатор, и для того, чтоб быстрее увидеть, как синтаксический анализатор взаимодействует со сканером, заменим пустую заглушку, использовавшуюся в роли процедуры `Compile` модуля `OPars` (см. листинг 3.3) содержательной заглушкой (листинг 3.13). Она будет читать лексемы до исчерпания входного текста, то есть до тех пор, пока очередной лексемой не станет `lexEOT`. Можно даже поручить процедуре `Compile` подсчет числа прочитанных лексем.

Листинг 3.13. Промежуточная версия синтаксического анализатора с подсчетом лексем

```
unit OPars;
{Распознаватель}
interface

procedure Compile;

{=====================================================}

implementation

uses OScan;

procedure Compile;
var
   n : integer;
begin
   n := 0;
   while Lex <> lexEOT do begin
      n := n + 1;
      NextLex;
   end;
   Writeln('Число лексем ', n );
end;

end.
```

Подсчет числа лексем интересен сам по себе. Количество лексем может служить мерой объема исходного текста программы. Ведь на размер программного текста в строках или байтах влияют многие вещи, которые не имеют отношения к содержанию программы, а зависят от индивидуального стиля программиста и особенностей языка программирования. Размер в байтах меняется в зависимости от длины имен, выбираемых программистом, длины служебных слов языка, количества пробелов. Число строк тоже зависит от индивидуальных привычек. Например, в программах, написанных Н. Виртом, можно видеть длинные строки, содержащие по несколько операторов, в то время как во многих других источниках, в том числе в этой книге, культивируется стиль, предполагающий запись не более одного оператора в строке.

Измерение числа лексем позволяет объективней оценить объем программистской работы и размеры программ.

Получившаяся у нас промежуточная версия компилятора способна обработать любую правильную программу на языке «О» и может сообщить о лексической ошибке и нарушении ограничений реализации при записи идентификаторов и чисел. В то же время наш анализатор будет без возражений обрабатывать текст, представляющий собой синтаксически неправильную последовательность правильно записанных лексем. Например, два приведенных в таблице 3.2 текста никаких ошибок не вызовут.

Таблица 3.2. Последовательности лексем

Правильная программа	Правильные лексемы
`MODULE Module;` `END Module.`	`.Module END` `;Module MODULE`

Синтаксический анализатор

Имея в распоряжении сканер, будем программировать синтаксический анализатор методом рекурсивного спуска, считая лексемы терминальными символами.

Структура распознающих процедур определяется синтаксисом соответствующих конструкций. Мы не будем пользоваться для задания синтаксиса диаграммами, поскольку формулы, записанные на РБНФ, и так достаточно наглядны. В тексте распознавателя перед каждой распознающей процедурой в качестве комментария будем записывать РБНФ-выражение, определяющее синтаксис соответствующей конструкции.

Все процедуры синтаксического анализатора располагаются в секции реализации модуля `OPars`. Интерфейс модуля (см. листинги 3.3 и 3.13) при этом не изменяется.

Начальным нетерминалом грамматики языка «О» является «Модуль». Соответствующая процедура распознавателя будет называться `Module`, а ее вызовом из процедуры `Compile` начнется синтаксический анализ:

```
procedure Compile;
begin
   Module;
   Writeln('Компиляция завершена');
end;
```

Находившуюся на этом месте заглушку, считавшую число лексем, пришлось удалить.

Процедура `Module` записывается в соответствии с синтаксисом, заданным формулой для нетерминала «Модуль».

```
 (* MODULE Имя ";" [Импорт] ПослОбъявл
    [BEGIN ПослОператоров] END Имя "." *)
procedure Module;
begin
   if Lex = lexMODULE then {Слово MODULE   }
      NextLex
   else
      Expected('MODULE');
   if Lex = lexName then   {Имя модуля     }
      NextLex
   else
      Expected('имя модуля');
   if Lex = lexSemi then   {Точка с запятой}
…
```

Можно было бы и дальше писать в таком стиле, но уже в самом начале мы столкнулись с тремя идущими подряд проверками соответствия текущей и ожидаемой лексемы. Проверки эти однотипны, и есть смысл поручить их выполнение специальной процедуре. Тем более, что и в дальнейшем подобные ситуации будут встречаться. Назовем такую процедуру `Check` (проверка, контроль). Её параметрами будут вид ожидаемой анализатором лексемы `L` и строка `M` (от message — «сообщение»), которая будет передана процедуре `Expected` и вставлена ею после слова «Ожидается» в сообщение об ошибке.

```
procedure Check(L: tLex; M: string);
begin
   if Lex <> L then
      Expected(M)
   else
      NextLex;
end;
```

Теперь анализатор модуля запишется проще (листинг 3.14).

Листинг 3.14. Синтаксический анализатор модуля

```
 (* MODULE Имя ";" [Импорт] ПослОбъявл
    [BEGIN ПослОператоров] END Имя "." *)
procedure Module;
begin
```

```
    Check(lexMODULE, 'MODULE');
    Check(lexName, 'имя модуля');
    Check(lexSemi, '";"');
    if Lex = lexIMPORT then
       Import;
    DeclSeq;    {Последовательность объявлений}
    if Lex = lexBEGIN then begin
       NextLex;
       StatSeq; {Последовательность операторов}
    end;
    Check(lexEND, 'END');
    Check(lexName, 'имя модуля');
    Check(lexDot, '"."');
 end;
```

Эта процедура выполняет синтаксический анализ в чистом виде, не пытаясь делать какие-либо дополнительные контекстные проверки. Так, не контролируется соответствие имени, записанного после слова **END** в конце модуля и названия модуля, идущего вслед за **MODULE**. Когда компилятор будет дополнен средствами контекстного анализа, такие проверки, конечно, будут выполняться. Приведенную же версию процедуры `Module` следует рассматривать как предварительную, в дальнейшем она будет дорабатываться.

Продолжим программирование анализатора, реализуя распознающие процедуры, обращение к которым уже предусмотрено процедурой `Module`. Первая в очереди — процедура-анализатор списка импорта (листинг 3.15). При ее рассмотрении также надо иметь в виду, что это чистый синтаксический анализ. Никаких попыток выполнить импорт фактически и даже проверить, действительно ли встречающиеся имена — это имена модулей, пока не предпринимается.

Листинг 3.15. Синтаксический анализатор списка импорта

```
 (* IMPORT Имя {"," Имя} ";" *)
 procedure Import;
 begin
    Check(lexIMPORT, 'IMPORT');
    Check(lexName, 'имя модуля');
    while Lex = lexComma do begin
       NextLex;
       Check(lexName, 'имя модуля');
    end;
    Check(lexSemi, '";"');
 end;
```

Далее (листинг 3.16) следует анализатор последовательности объявлений.

Листинг 3.16. Распознающая процедура для последовательности объявлений

```
 (* {CONST {ОбъявлКонст ";"}
     |VAR {ОбъявлПерем ";"} } *)
procedure DeclSeq;
begin
   while Lex in [lexCONST, lexVAR] do begin
      if Lex = lexCONST then begin
         NextLex;
         while Lex = lexName do begin
            ConstDecl; {Объявление константы}
            Check(lexSemi, '";"');
         end;
         end
      else begin
         NextLex; { VAR }
         while Lex = lexName do begin
            VarDecl;   {Объявление переменных}
            Check(lexSemi, '";"');
         end;
      end;
   end;
end;
```

Это окончательный текст процедуры `DeclSeq` (от declarations sequence — последовательность объявлений). Никаких контекстных проверок и действий по генерации машинного кода на этом уровне анализа происходить не будет, поскольку любые такие действия относятся к объявлениям конкретных констант и переменных, а их обработка будет сконцентрирована в процедурах `ConstDecl` и `VarDecl`.

Анализатор последовательности операторов (листинг 3.17) также в дальнейшем не изменится.

Листинг 3.17. Анализатор последовательности операторов

```
 (* Оператор {";" Оператор} *)
procedure StatSeq;
begin
   Statement;    {Оператор}
   while Lex = lexSemi do begin
      NextLex;
```

```
      Statement; {Оператор}
   end;
end;
```

С нетерминалом «последовательность операторов» в грамматике языка «О» связано самовложение. Последовательность операторов состоит из операторов, а некоторые операторы содержат в себе последовательности операторов. В программе-анализаторе это порождает косвенную рекурсию. Чтобы заголовки участвующих в рекурсии процедур были известны до места вызова этих процедур, предусмотрим опережающее описание процедуры `StatSeq`, которое можно разместить в начале секции реализации модуля `OPars`:

```
procedure StatSeq; forward;
```

Цепочку распознающих процедур можно было бы разворачивать и дальше. Реализуя алгоритм рекурсивного спуска, это можно сделать легко и быстро. В ходе реальной разработки так и следует поступить, завершив синтаксический анализатор полностью. Но здесь и сейчас я не буду выписывать все распознающие процедуры, поскольку большинство из них подвергнутся модернизации при создании контекстного анализатора и генератора кода. Тогда их и запишем. Принцип же, надеюсь, ясен. Приведу для примера лишь процедуру, анализирующую слагаемое (листинг 3.18), — она является частью анализатора выражений.

Листинг 3.18. Синтаксический анализатор слагаемого

```
 (* Множитель {ОперУмн Множитель} *)
procedure Term; {Слагаемое}
begin
   Factor;    {Множитель}
   while Lex in [lexMult, lexDIV, lexMOD] do begin
      NextLex;
      Factor; {Множитель}
   end;
end;
```

Контекстный анализ

В ходе контекстного анализа должно быть проверено соблюдение тех правил языка, которые не выражаются с помощью контекстно-

свободных грамматик[71]. Примерами являются проверка правильности употребления имен и контроль соответствия типов.

Таблица имен

Корректность использования имени в конкретном месте программы может быть проверена, если известно, какой объект программы этим именем обозначен: константа, тип, переменная, процедура, модуль. Для имени константы, переменной, процедуры (функции) необходимо также знать, к какому типу они относятся. Могут быть важны и другие характеристики, связанные с именем.

Атрибуты каждого имени, используемого в программе, хранятся в специальной таблице транслятора — таблице имен. Заполняется таблица имен при трансляции объявлений и списка импорта. Предопределенные имена (имена стандартных типов, процедур) могут быть занесены в таблицу заранее. При трансляции операторов обращение к таблице имен позволяет определить атрибуты каждого встретившегося имени, и служит для выявления необъявленных имен.

Блочная структура и области видимости

Структура таблицы имен должна отражать блочную структуру программы. Одно и то же имя может обозначать несколько объектов в программе, если они определены в разных ее блоках. Поэтому сведения об именах также надо хранить в разных блоках таблицы имен. И хотя в языке «О» понятие «блок» не используется, мы учтем блочную структуру программы при организации таблицы имен. Во-первых, это будет полезно при последующем расширении языка, во-вторых, как скоро выяснится, пригодится уже при существующем его

[71] Точнее сказать, не выражены с помощью КС-грамматики авторами спецификации языка. В некоторых случаях, требования, которые могут быть в принципе заданы с помощью синтаксических правил, разумнее формулировать словесно, чтобы не усложнять формальную грамматику. Примером может служить проблема «висячего else» в языках Паскаль, Си, Ява, Си#, связанная с потенциально неоднозначной трактовкой конструкции, содержащей последовательность `then if` (в Си-подобных языках: if(...) if ...). В спецификации Паскаля и Си# просто замечено, что `else` всегда относится к ближайшему предшествующему `if`, в то время как в спецификации языка Ява это выражено с помощью правил КС-грамматики, что заметно ее усложнило.

состоянии, в-третьих, позволит представить организацию таблицы имен в трансляторах «настоящих» языков.

В языке Оберон блоки образуются модулями и процедурами. Рассмотрим программу на Обероне (листинг 3.19) — модуль, содержащий вложенные процедуры.

Листинг 3.19. Блочная структура и области видимости

```
MODULE M;
VAR
   v1, v2, v3 : INTEGER;
PROCEDURE P1(…);
VAR
   v1, v2 : INTEGER;
   PROCEDURE P11(…);
   VAR
      v1 : INTEGER;
   BEGIN
      {Здесь видны локальная переменная v1,
       переменная v2 процедуры P1
       а также глобальная переменная v3.
       Видны также имена P11, P1 и M}
   …
   END P11;
BEGIN
   {Здесь видны P11, P1, M,
    локальные v1 и v2 процедуры P1,
    а также глобальная переменная v3}
   …
END P1;
PROCEDURE P2(…);
VAR
   v2 : INTEGER;
BEGIN
   {Здесь видны P2, P1, M,
    локальная переменная v2 процедуры P2,
    а также глобальные v1 и v3}
   …
END P2;
BEGIN
   {Здесь видны P2, P1, M и глобальные v1, v2, v3}
   …
END M.
```

Область видимости[72] имени простирается от точки его объявления до конца блока (модуля, процедуры), в начале которого это объявление находится. Из нее исключаются области видимости одноименных объектов, объявленных во вложенных блоках.

Правильно считать «точкой объявления» имени то место в программе, где это имя записано, а не конец объявления, в котором это имя содержится. Например, началом области видимости переменной v1, содержащейся в объявлении

```
VAR
    v1{точка 1}, v2, v3: INTEGER;{точка 2}
```

следует считать точку 1, а не точку 2.

Блоки программы можно рассматривать как *пространства имен* — пространство имен модуля, пространство имен процедуры. Области видимости отдельных идентификаторов вложены в соответствующие пространства имен.

Блоки таблицы имен создаются компилятором при обработке начала каждого блока программы (модуля, процедуры). После того, как однопроходный компилятор закончил трансляцию процедурного блока, соответствующий блок таблицы имен может быть уничтожен, ведь области видимости локальных имен этой процедуры закончились, и эти имена не могут быть видны в последующих частях программы.

Последовательность создания и уничтожения блоков таблицы имен при трансляции программы, приведенной в листинге 3.19, получается такой.

1. Создание блока для пространства имен модуля (блок M).
2. Создание блока для процедуры P1 (блок P1) при входе в процедуру P1, то есть при трансляции заголовка процедуры P1.
3. Создание блока для процедуры P11 (блок P11).
4. Уничтожение блока P11 по окончании трансляции процедуры P11.
5. Уничтожение блока P1.
6. Создание блока для процедуры P2 (блок P2).
7. Уничтожение блока P2.
8. Уничтожение блока M по окончании трансляции модуля.

[72] Используется также название «область действия». Соответствующий английский термин — scope.

Можно видеть, что создание и уничтожение блоков таблицы имен подчиняется стековой дисциплине: блок, созданный последним, уничтожается первым. Соответственно, и в таблице имен блоки должны образовывать стек.

Блок стандартных идентификаторов

Кроме имен, которые программист определяет сам, в программе могут использоваться предопределенные идентификаторы: имена стандартных типов, процедур, функций. В Обероне (и языке «О») такими именами являются, например, `INTEGER`, `ABS`, `MAX`. Их область видимости распространяется на весь текст модуля. Но такие имена не являются зарезервированными словами (в отличие, например, от **`BEGIN`**), и программист может придать им в программе другой смысл. Если записать

```
VAR MAX: INTEGER;
```

то в остальной части блока, в котором находится это объявление, `MAX` будет обозначать переменную целого типа, а не стандартную функцию `MAX`.

Таким образом, по отношению к стандартным идентификаторам могут быть применены обычные правила, относящиеся к пространствам имен, если считать, что эти идентификаторы определены в блоке, охватывающем модуль. Блок стандартных идентификаторов открывается в таблице имен перед началом трансляции модуля[73]. Стандартные идентификаторы заносятся в таблицу, после чего может быть открыт блок для пространства имен модуля.

В связи с необходимостью правильной обработки стандартных идентификаторов блочная структура таблицы имен оказывается актуальной и для компилятора языка «О».

Таблица имен компилятора «О»

Выберем простой, хоть и не очень эффективный (из-за медленного поиска имени) способ организации таблицы имен — линейный список. Если добавление и удаление элементов выполнять с одного конца списка, он ведет себя подобно стеку, что и требуется для таблицы имен. Чтобы разграничить блоки таблицы, используем элементы, которые в одном из своих полей будут содержать специальный признак (рис. 3.5).

[73] В языке «О» модуль и программа — это одно и то же.

Рис. 3.5. Устройство таблицы имен компилятора языка «О»

Информация в таблице имен

Для каждого имени, встретившегося в программе на языке «О», в таблице имен будет храниться следующая информация:

- Само имя в виде строки символов. Будет служить ключом при поиске.
- Категория имени. Что имя обозначает: константу, переменную, тип, стандартную процедуру, модуль.
- Тип. Должен быть указан для имен переменных, констант, процедур-функций.
- Значение. Для имени константы это будет ее числовое значение. Для переменных и процедур это поле также пригодится.

Программный модуль OTable

Теперь можно определить программный интерфейс модуля `OTable` (листинг 3.20), который в нашем компиляторе будет отвечать за работу с таблицей имен.

Листинг 3.20. Интерфейс модуля для работы с таблицей имен

```
unit OTable;
{ Таблица имен }

interface
```

```
uses OScan;

type
   {Категории имён}
      tCat = (catConst, catVar, catType,
              catStProc, catModule, catGuard);

   {Типы}
      tType = (typVoid, typInt, typBool);

   tObj = ^tObjRec;  {Тип указателя на запись таблицы}
   tObjRec = record      {Тип записи таблицы имен}
      Name   : tName;    {Ключ поиска              }
      Cat    : tCat;     {Категория имени          }
      Typ    : tType;    {Тип                      }
      Val    : integer;  {Значение                 }
      Prev   : tObj;     {Указатель на пред. имя }
   end;

{Инициализация таблицы}
   procedure InitNameTable;
{Добавление элемента}
   procedure Enter
      (N: tName; C: tCat; T: tType; V: integer);
{Занесение нового имени}
   procedure NewName
      (Name: tName; Cat: tCat; var Obj: tObj);
{Поиск имени}
   procedure Find(Name: tName; var Obj: tObj);
{Открытие области видимости (блока)}
   procedure OpenScope;
{Закрытие области видимости (блока)}
   procedure CloseScope;

{====================================================}
```

Среди значений перечислимого типа `tCat`, обозначающего категории объектов в таблице имен, предусмотрена константа `catGuard`, которая будет обозначать специальный элемент таблицы, разграничивающий ее блоки (guard по-английски — страж, ограждение; frontier guard — пограничник). Значение `typVoid`, использованное в определении `tType`, потребуется для указания об отсутствии типа

у объекта программы. Например, стандартные процедуры (не функции) не имеют типа, чем и отличаются от процедур-функций.

Инициализация таблицы, организованной как линейный список, может быть сведена к установке в значение **nil** указателя на список. Переменную, обозначающую указатель на начало (вершину) списка, как и указатель на его конец (дно) определим как глобальные в секции реализации модуля `OTable`:

```
var
   Top    : tObj;  {Указатель на вершину списка    }
   Bottom : tObj;  {Указатель на конец (дно) списка}

{Инициализация таблицы имен}
procedure InitNameTable;
begin
   Top := nil;
end;
```

Добавление элемента к списку будет выполнять процедура `Enter` (листинг 3.21). Ее параметрами являются значения полей записи таблицы имен.

Листинг 3.21. Добавление элемента в таблицу имен

```
procedure Enter(N:tName; C:tCat; T:tType; V:integer);
var
   P : tObj;
begin
   New(P);
   P^.Name := N;
   P^.Cat := C;
   P^.Typ := T;
   P^.Val := V;
   P^.Prev := Top;
   Top := P;
end;
```

Процедуры `OpenScope` и `CloseScope` (листинг 3.22) отвечают за открытие и закрытие (уничтожение) блоков таблицы имен, соответствующих пространствам имен в программе.

Листинг 3.22. Открытие и закрытие областей видимости

```
procedure OpenScope;
begin
   Enter('', catGuard, typVoid, 0);
   if Top^.Prev = nil then
```

```
        Bottom := Top;
end;

procedure CloseScope;
var
   P : tObj;
begin
   while Top^.Cat <> catGuard do begin
      P := Top;
      Top := Top^.Prev;
      Dispose(P);
   end;
   P := Top;
   Top := Top^.Prev;
   Dispose(P);
end;
```

Процедура `OpenScope` помещает в список элемент, разделяющий блоки таблицы. Этот элемент содержит значение `catGuard` в поле категории имени. Если открывается первый блок таблицы имен, указатель `Bottom` устанавливается на добавленный пограничный элемент. Указатель `Bottom` будет использоваться при поиске имен.

Процедура `CloseScope` удаляет блок таблицы имен, находящийся на вершине стека (открытый последним). Для этого уничтожаются все записи от вершины списка до ближайшего пограничного элемента включительно.

Заполнение таблицы имен

Происходит при трансляции объявлений, которые могут располагаться в начале каждого блока программы. Когда транслятор встречает объявление имени, он должен занести его и сопутствующую информацию в тот блок таблицы имен, который соответствует текущему блоку программы. Блок, в который помещаются данные, является последним по времени открытым блоком таблицы. Перед занесением необходимо проверить, нет ли уже в этом последнем блоке такого же имени. По правилам Оберона, а вслед за ним языка «О», как и по правилам многих других языков, один идентификатор не может быть объявлен в одном блоке дважды.

Добавление в таблицу нового имени `Name` будет выполнять процедура `NewName` (листинг 3.23). Входной параметр `Cat` определяет категорию имени. Указатель `Obj` на созданную в таблице запись возвра-

щается как результат работы NewName. Другие атрибуты помещенного в таблицу идентификатора (тип обозначаемого объекта, значение), могут быть сформированы вызывающей программой уже после вызова NewName. Значение Val при инициализации устанавливается равным 0, что будет использовано в дальнейшем.

Перед записью имени в таблицу проверяется, нет ли **в пределах текущего блока** такого же. Если обнаружено совпадение, сообщается об ошибке.

Листинг 3.23. Запись нового имени в таблицу имен

```
procedure NewName(Name:tName; Cat:tCat; var Obj:tObj);
begin
   Obj := Top;
   while (Obj^.Cat<>catGuard) and (Obj^.Name<>Name) do
      Obj := Obj^.Prev;
   if Obj^.Cat = catGuard then begin
      New(Obj);
      Obj^.Name := Name;
      Obj^.Cat := Cat;
      Obj^.Val := 0;
      Obj^.Prev := Top;
      Top := Obj;
      end
   else
      Error('Повторное объявление имени');
end;
```

Поиск имен

Занесение нового имени в таблицу происходит при обработке его *определяющего вхождения*, то есть когда имя объявляется. При трансляции *использующего вхождения* выполняется поиск имени в таблице. По правилам Оберона любому использующему вхождению должно предшествовать определяющее. Исключение составляют стандартные идентификаторы, которые являются предопределенными и, как мы уже решили, считаются описанными во внешнем по отношению к модулю блоке.

Процедура Find (листинг 3.24) ищет в таблице имя Name. Поиск начинается с вершины списка и **не останавливается на границах блоков**. Если во вложенных блоках программы объявлены одинаковые имена, найдено будет то, которое определено в ближайшем внут-

реннем блоке. И только если имя не объявлено в текущем блоке и ни в одном из охватывающих его, сообщается об ошибке.

Листинг 3.24. Поиск имени

```
procedure Find(Name: tName; var Obj: tObj);
begin
   Bottom^.Name := Name;
   Obj := Top;
   while Obj^.Name <> Name do
      Obj := Obj^.Prev;
   if Obj = Bottom then
      Error('Необъявленное имя');
end;
```

Результатом поиска является ссылка `Obj`, указывающая на запись о найденном имени. С помощью этой ссылки можно получить всю информацию об объекте, обозначенном этим именем.

При поиске используется «барьер» — перед началом поиска в поле `Name` последнего в списке элемента (на который указывает `Bottom`), помещается искомый ключ.

Напомню, что процедура `Error`, которая может быть вызвана при неудачной попытке занесения имени в таблицу и при неудачном поиске, останавливает работу всего компилятора.

Контекстный анализ модуля

Контекстный анализатор не является самостоятельным блоком компилятора. Действия, связанные с проверкой имен и соответствия типов, встраиваются в синтаксический анализатор. Написанные раньше части распознавателя должны быть модернизированы с целью добавления в них действий по контекстному анализу. Первой изменим основную процедуру анализатора — `Compile` (листинг 3.25). В ее задачу теперь будет входить инициализация таблицы имен, открытие в таблице блока стандартных идентификаторов, занесение в этот блок стандартных имен, открытие блока, соответствующего области видимости идентификаторов модуля, вызов, как и прежде, распознавателя `Module` и закрытие обоих открытых блоков таблицы имен.

Листинг 3.25. Основная процедура распознавателя

```
procedure Compile;
begin
   InitNameTable;
   OpenScope; {Блок стандартных имен}
```

```
    Enter( 'ABS', catStProc, typInt, spABS );
    Enter( 'MAX', catStProc, typInt, spMAX );
    Enter( 'MIN', catStProc, typInt, spMIN );
    Enter( 'DEC', catStProc, typVoid, spDEC );
    Enter( 'ODD', catStProc, typBool, spODD );
    Enter( 'HALT', catStProc, typVoid, spHALT );
    Enter( 'INC', catStProc, typVoid, spINC );
    Enter( 'INTEGER', catType, typInt, 0 );
    OpenScope;  {Блок модуля}
    Module;
    CloseScope; {Блок модуля}
    CloseScope; {Блок стандартных имен}
    WriteLn;
    WriteLn('Компиляция завершена');
end;
```

Занесение в таблицу стандартных идентификаторов выполняется с помощью процедуры `Enter`. Для имен стандартных процедур в качестве значений заносятся их условные номера, которые в дальнейшем будут использованы при обработке вызовов процедур. Определены эти номера могут быть в секции реализации модуля `OPars`:

```
const
    spABS     = 1;
    spMAX     = 2;
    spMIN     = 3;
    spDEC     = 4;
    spODD     = 5;
    spHALT    = 6;
    spINC     = 7;
    spInOpen  = 8;
    spInInt   = 9;
    spOutInt  = 10;
    spOutLn   = 11;
```

Для процедур-функций указывается их тип. В поле типа для `INC` и `DEC`, которые функциями не являются, заносится `typVoid`. Значение для идентификатора `INTEGER` не требуется, а в соответствующее поле просто записывается ноль.

Обозначения процедур ввода-вывода при инициализации таблицы имен в нее не заносятся. Хотя можно видеть, что константы, задающие их условные номера, (`spInOpen` – `spOutLn`) уже определены. Идентификаторы из стандартных модулей `In` и `Out` будут занесены в таблицу имен при импорте соответствующего модуля.

Первая возможность продемонстрировать работу с таблицей имен предоставляется при модернизации распознающей процедуры `Module` (листинг 3.26). Если раньше синтаксический анализатор (см. листинг 3.13) лишь убеждался в наличии какого-либо имени после слова **MODULE** в начале и после слова **END** в конце, то в ходе контекстного анализа нужно проверить, одинаковы ли эти имена. Для этого идентификатор, записанный после слова **MODULE,** заносится в таблицу, а имя, встретившееся после **END**, сравнивается с именем из таблицы. Для сохранения ссылки на запись об имени модуля используется локальная переменная `ModRef` (от Module Reference — ссылка на модуль).

Листинг 3.26. Распознаватель модуля

```
 (* MODULE Имя ";" [Импорт] ПослОбъявл
    [BEGIN ПослОператоров] END Имя "." *)
procedure Module;
var
   ModRef: tObj; {Ссылка на имя модуля в таблице}
begin
   Check(lexMODULE, 'MODULE');
   if Lex <> lexName then
      Expected('имя модуля')
   else {Имя модуля - в таблицу имен}
      NewName(Name, catModule, ModRef);
   NextLex;
   Check(lexSemi, '";"');
   if Lex = lexIMPORT then
      Import;
   DeclSeq;
   if Lex = lexBEGIN then begin
      NextLex;
      StatSeq;
   end;
   Check(lexEND, 'END');

   {Сравнение имени модуля и имени после END}
      if Lex <> lexName then
         Expected('имя модуля')
      else if Name <> ModRef^.Name then
         Expected('имя модуля "' + ModRef^.Name + '"')
      else
         NextLex;
   Check(lexDot, '"."');
end;
```

Конечно, сравнить имя, записанное после **MODULE**, с именем после соответствующего **END** можно и не обращаясь к таблице имен. Достаточно в начале запомнить (в локальной переменной) строковое значение переменной сканера Name, а затем сравнить его с именем, обнаруженным после **END**. Однако два этих решения не эквивалентны. Занесение имени модуля в таблицу означает, что в последовательности объявлений модуля уже нельзя будет определить другой объект с тем же именем. Если же имя модуля в таблицу не заносится, то такое становится возможным. Другой нюанс: если имя модуля занесено в таблицу, оно может перекрыть видимость одного из стандартных идентификаторов. Например, назвав модуль INTEGER, мы сделаем невозможным использование стандартного типа INTEGER в последовательности объявлений этого модуля.

Которое из двух решений правильное — вопрос совсем не однозначный. Имя модуля должно быть «видно» в той среде, где модуль используется. Это может быть среда Оберон-системы или операционная система. В этом смысле имя модуля не должно быть локальным внутри самого модуля[74], и решение не заносить его в таблицу имен вполне правомерно. С другой стороны, по крайней мере, в языке «О», где программа состоит из единственного модуля, нет причин разрешать применение имени модуля в каком-либо ином смысле.

Трансляция списка импорта

Синтаксический анализатор списка импорта приведен выше в листинге 3.14. Однако этот анализатор лишь убеждается, что в списке имеется одно или больше имен, не проверяя, действительно ли это имена существующих модулей, нет ли в списке повторов и попытки импортировать в модуле его самого. Действия по трансляции импорта отдельного модуля поручим специальной процедуре ImportModule. Распознаватель списка импорта, вызывающий ImportModule, приведен в листинге 3.27.

Листинг 3.27. Анализатор списка импорта

```
(* IMPORT Имя { "," Имя } ";" *)
procedure Import;
begin
```

[74] Немного забегая вперед, можно заметить, что именно так обстоит дело с именами процедур: они относятся к блоку, в котором располагается сама процедура, а не к блоку процедуры.

```
    Check(lexIMPORT, 'IMPORT');
    ImportModule;    {Импорт модуля}
    while Lex = lexComma do begin
       NextLex;
       ImportModule; {Импорт модуля}
    end;
    Check(lexSemi, '";"');
end;
```

Процедура ImportModule (листинг 3.28), убедившись, что текущая лексема — имя, заносит его в таблицу для того, чтобы это имя было видно в оставшейся части компилируемой программы. При добавлении в таблицу будет проверено, что имя не упомянуто дважды. В частности, при попытке импортировать модулем самого себя также будет сообщено о повторном объявлении, поскольку имя компилируемого модуля уже занесено в тот же блок таблицы.

Листинг 3.28. Импорт модуля

```
procedure ImportModule;
var
  ImpRef: tObj;
begin
  if Lex = lexName then begin
    NewName(Name, catModule, ImpRef);
    if Name = 'In' then begin
       Enter('In.Open', catStProc, typVoid, spInOpen);
       Enter('In.Int', catStProc, typVoid, spInInt);
       end
    else if Name = 'Out' then begin
       Enter('Out.Int', catStProc, typVoid, spOutInt);
       Enter('Out.Ln', catStProc, typVoid, spOutLn);
      end
    else
       Error('Неизвестный модуль');
    NextLex;
    end
  else
     Expected('имя импортируемого модуля');
end;
```

После того, как имя импортируемого модуля занесено в таблицу, выполняется собственно импорт.

При трансляции с «настоящего» языка Оберон это означало бы поиск файла модуля или файла спецификации его интерфейса в среде

Оберон-системы. В упрощенном учебном компиляторе языка «О» предусмотрены лишь два стандартных модуля `In` и `Out`. Они, по существу, встроены в язык. Никаких внешних файлов, в которых хранится код или спецификация этих модулей не предусматривается. Компилятор «знает» про существование модулей `In` и `Out`. При их импорте в таблицу добавляются заранее известные компилятору имена экспортированных этими модулями процедур. Обозначения процедур `In.Open`, `In.Int`, `Out.Int` и `Out.Ln` заносятся в текущий блок таблицы имен вместе с именем модуля и точкой. Это позволяет в программе на языке «О» обратиться к таким процедурам только по их уточненным (квалифицированным) именам, включающим имя модуля, и только при условии, что соответствующий модуль программой импортирован.

Трансляция описаний

Распознаватель последовательности объявлений можно видеть выше в листинге 3.16. Никаких изменений, связанных с контекстным анализом, вносить в процедуру `DeclSeq` из листинга 3.16 не требуется, поскольку она отвечает лишь за контроль синтаксиса в последовательности объявлений констант и переменных, не имея дела с конкретными переменными и константами. Вся работа по контекстному анализу выполняется процедурой `ConstDecl`, обрабатывающей определение отдельной константы и процедурой `VarDecl`, ответственной за одно описание переменных.

Трансляция определений констант

Начнем с трансляции объявлений констант (листинг 3.29). Как и всегда, встретив определяющее вхождение имени, транслятор заносит его в таблицу, сопровождая сведениями о том, к какой категории это имя относится (при вызове `ConstDecl` имя уже является текущей лексемой). В нашем случае речь идет об имени константы (категория `catConst`), но при вызове `NewName` укажем категорию `catGuard`. Делается это для того, чтобы предотвратить тавтологию — определение константы через саму себя. Очевидно, что конструкции вроде

```
CONST C = C;
```

должны быть запрещены. Если же в момент записи в таблицу имен сразу назначить определяемому имени категорию константы, его тут же можно будет использовать в этой роли. Значение `catConst` зане-

сем в поле категории лишь после того, как константа будет определена полностью.

Листинг 3.29. Трансляция объявления константы

```
 (* Имя "=" КонстВыраж *)
procedure ConstDecl;
var
   ConstRef: tObj; {Ссылка на имя в таблице}
begin
   NewName(Name, catGuard, ConstRef);
   NextLex;
   Check(lexEQ, '"="');
   ConstExpr(ConstRef^.Val);
   ConstRef^.Typ := typInt;{Констант других типов нет}
   ConstRef^.Cat := catConst;
end;
```

Для анализа и вычисления константного выражения, которое записывается в определении константы после знака «=», будет служить процедура `ConstExpr`. Ее выходным параметром является числовое значение выражения. В качестве фактического параметра при вызове `ConstExpr` подставим поле значения (поле `Val`) той записи таблицы имен, где хранятся сведения об определяемой константе. Эта запись доступна через указатель `ConstRef`.

Напомню, что в языке «О» определены константные выражения лишь специального вида. В константном выражении можно использовать число или имя константы (со знаком или без него). Приведенная в листинге 3.30 процедура `ConstExpr` распознает константное выражение и вычисляет его, присваивая найденное значение своему выходному параметру `V`.

Листинг 3.30. Анализ и вычисление константного выражения

```
 (* ["+" | "-"] (Число | Имя) *)
procedure ConstExpr(var V: integer);
var
   X  : tObj;
   Op : tLex;
begin
   Op := lexPlus;
   if Lex in [lexPlus, lexMinus] then begin
      Op := Lex;
      NextLex;
   end;
```

```
   if Lex = lexNum then begin
      V := Num;
      NextLex;
      end
   else if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catGuard then
         Error(
            'Нельзя определять константу через себя'
         )
      else if X^.Cat <> catConst then
         Expected('имя константы')
      else
         V := X^.Val;
      NextLex;
      end
   else
      Expected( 'константное выражение' );
   if Op = lexMinus then
      V := -V;
end;
```

Трансляция описаний переменных

Задача процедуры VarDecl (листинг 3.31), отвечающей за трансляцию одного описания переменных (список переменных, за которым следует тип) довольно проста. Имена переменных должны быть занесены в таблицу со значением атрибута «категория» равным catVar и снабжены указанием об их типе. Дело облегчается тем, что в языке «О» существуют лишь переменные типа INTEGER, поэтому значение поля Typ в записи об имени можно заполнить сразу, даже до распознавания самого типа.

Листинг 3.31. Трансляция описания переменных

```
(* Имя {"," Имя} ":" Тип *)
procedure VarDecl;
var
   NameRef : tObj;
begin
   if Lex <> lexName then
      Expected('имя')
   else begin
      NewName(Name, catVar, NameRef);
      NameRef^.Typ := typInt;
```

```
         NextLex;
      end;
      while Lex = lexComma do begin
         NextLex;
         if Lex <> lexName then
            Expected('имя')
         else begin
            NewName(Name, catVar, NameRef);
            NameRef^.Typ := typInt;
            NextLex;
         end;
      end;
      Check(lexColon, '":"');
      ParseType;
   end;
```

Распознающая процедура для типа названа `ParseType` (распознать тип), а не `Type`, поскольку **`type`** в языке Паскаль зарезервировано.

Можно обратить внимание, что в записях об именах переменных осталось незаполненным поле значения (поле `Val`). Пока у нас нет для этого необходимой информации. Она появится лишь при рассмотрении генерации машинного кода.

Трансляция объявлений констант и переменных связана лишь с заполнением таблицы имен и не порождает никакого машинного кода.

Контекстный анализ выражений

Выражения — важнейший элемент любого языка программирования. В языке «О» предусмотрены арифметические (типа `INTEGER`) и логические выражения. Последние могут использоваться только в операторах **`IF`** и **`WHILE`**. Синтаксис выражений языка «О» определяется следующими РБНФ-формулами:

```
Выраж = ПростоеВыраж [Отношение ПростоеВыраж].
ПростоеВыраж = ["+"|"-"] Слагаемое
   {ОперСлож Слагаемое}.
Слагаемое = Множитель {ОперУмн Множитель}.
Множитель = Имя ["(" Выраж | Тип ")"]
   | Число | "(" Выраж ")".
```

В задачу контекстного анализатора при трансляции выражений входит проверка соответствия операций и типов операндов. Поскольку синтаксис выражений иерархичен (выражение → простое выражение → слагаемое → множитель), распознающие процедуры нижнего

уровня должны сообщать процедурам верхнего уровня тип соответствующего подвыражения. Каждый распознаватель в иерархии выражений должен определять (скажем даже «вычислять») тип соответствующего подвыражения.

Итак, распознаватели, участвующие в анализе выражений, проверяют соответствие типов операндов и операций и *вычисляют типы* подвыражений. Ни о какой генерации кода и вычислении значений выражений речь пока не идет — «вычисляются» лишь типы выражений.

Каждый распознаватель снабдим выходным параметром (**var** T: tType), обозначающим тип подвыражения. Вначале программируем (листинг 3.32) процедуру Expression — выражение.

Листинг 3.32. Распознаватель выражений

```
 (* ПростоеВыраж [Отношение ПростоеВыраж] *)
procedure Expression(var T : tType);
begin
  SimpleExpr(T); {Получить тип первого подвыражения}
  if Lex in [lexEQ, lexNE, lexGT, lexGE, lexLT, lexLE]
  then begin
    if T <> typInt then
       Error('Несоответствие операции типу операнда');
    NextLex;
    SimpleExpr(T); {Правый операнд отношения}
    if T <> typInt then
       Expected('выражение целого типа');
    T := typBool;
  end;{иначе тип равен типу первого прост. выражения}
end;
```

Обратите внимание, что контроль типов происходит только в связи с распознаванием операции.

Далее по иерархии следует распознаватель простых выражений (листинг 3.33).

Листинг 3.33. Распознаватель простого выражения

```
 (* ["+"|"-"] Слагаемое {ОперСлож Слагаемое} *)
procedure SimpleExpr(var T : tType);
begin
   if Lex in [lexPlus, lexMinus] then begin
      NextLex;
      Term(T);
      if T <> typInt then
```

```
            Expected('выражение целого типа');
         end
      else
         Term(T);
      if Lex in [lexPlus, lexMinus] then begin
         if T <> typInt then
            Error
               ('Несоответствие операции типу операнда');
         repeat
            NextLex;
            Term(T);
            if T <> typInt then
               Expected('выражение целого типа');
         until not( Lex in [lexPlus, lexMinus] );
      end;
   end;
```

Как можно видеть, необходимость выполнения контекстных проверок несколько изменила реализацию распознавателя. Так, вместо цикла

```
while Lex in [lexPlus, lexMinus] do begin
   NextLex;
   Term
end;
```

который в синтаксическом анализаторе выполнял бы обработку второго и последующих слагаемых, использована конструкция из **`if`** и **`repeat`**, позволяющая вовремя проверить тип первого слагаемого.

За исключением случая, когда простое выражение состоит из одного слагаемого (терма), его тип будет целым. Если слагаемое одно, тип простого выражения совпадает с типом этого слагаемого.

Аналогично строится распознаватель слагаемого (листинг 3.34).

Листинг 3.34. Распознаватель слагаемого

```
 (* Множитель {ОперУмн Множитель} *)
procedure Term(var T: tType);
begin
   Factor(T);
   if Lex in [lexMult, lexDIV, lexMOD] then begin
      if T <> typInt then
         Error
            ('Несоответствие операции типу операнда');
      repeat
         NextLex;
```

```
            Factor(T);
            if T <> typInt then
               Expected('выражение целого типа');
         until not( Lex in [lexMult, lexDIV, lexMOD] );
      end;
   end;
```

Множитель представляет собой первичное выражение, элементарный операнд. В этой роли могут выступать переменная, именованная константа, вызов процедуры-функции, число, наконец, выражение в скобках. Распознаватель множителя (листинг 3.35) серией последовательных проверок отделяет эти варианты. Поскольку первые три вида множителя (переменная, константа, функция) начинаются с имени, для их разделения используется обращение к таблице имен. В зависимости от категории имени обрабатывается тот или иной вариант.

Листинг 3.35. Распознаватель множителя

```
(* Имя["(" Выраж | Тип ")"] | Число | "(" Выраж ")" *)
procedure Factor(var T: tType);
var
   X : tObj;
begin
   if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catVar then begin
         {Переменная}
            T := X^.Typ;
            NextLex;
         end
      else if X^.Cat = catConst then begin
         {Константа}
            T := X^.Typ;
            NextLex;
         end
      else if (X^.Cat=catStProc) and (X^.Typ<>typVoid)
      then begin
         {Процедура-функция}
            NextLex;
            Check(lexLPar, '"("');
            StFunc(X^.Val, T);
            Check(lexRPar, '")"');
         end
      else
```

```
            Expected(
            'переменная, константа или процедура-функция'
            );
         end
      else if Lex = lexNum then begin
         {Число}
            T := typInt;
            NextLex
         end
      else if Lex = lexLPar then begin
         {Выражение в скобках}
            NextLex;
            Expression(T);
            Check(lexRPar, '")"');
         end
      else
         Expected('имя, число или "("');
   end;
```

Обработку списка фактических параметров стандартной функции выполнит процедура `StFunc`. Она же вычислит тип множителя, представляющего собой вызов процедуры-функции. Заголовок этой `StFunc` будет таким:

```
procedure StFunc(F: integer; var T: tType);
```

Здесь `F` — номер функции (`stABS`, `stMAX`, …); `T` — выходной параметр — тип функции. Запрограммирована эта процедура будет позже.

Контекстный анализ операторов

В языке «О» имеется четыре вида операторов: присваивание, вызов процедуры, оператор **`if`** и оператор **`while`**. Как следует из синтаксической формулы (листинг 3.36), оператор также может быть пустым.

Листинг 3.36. Синтаксис оператора языка «О»

```
Оператор = [
   Переменная ":=" Выраж
   |[Имя "."] Имя ["(" [Параметр {"," Параметр}] ")"]
   |IF Выраж THEN
      ПослОператоров
   {ELSIF Выраж THEN
      ПослОператоров}
   [ELSE
      ПослОператоров]
    END
```

```
    |WHILE Выраж DO
       ПослОператоров
    END
 ].
```

Синтаксис всех видов операторов определяется единым РБНФ-правилом. Если буквально следовать технологии рекурсивного спуска, нужно записать одну распознающую процедуру, транслирующую все операторы. Такая процедура будет достаточно громоздкой. Предусмотрим свои распознающие процедуры для каждого из видов операторов, в то время как на распознаватель `Statement` (оператор) будет лишь возложена обязанность, определить, с каким из четырех случаев он имеет дело и вызвать соответствующую распознающую процедуру.

Первые два варианта (присваивание и вызов процедуры) не могут быть различены из анализа одной текущей лексемы: и тот и другой оператор начинается с имени. Распознавание может быть выполнено с привлечением контекстной информации. Если имя принадлежит переменной, то далее следует ожидать присваивание, иначе можно предположить обозначение процедуры — стандартной или из импортированного стандартного модуля.

Итак, если первая лексема оператора — имя, выполняем поиск в таблице имен с целью определения его категории:

```
    if Lex = lexName then begin
       Find(Name, X);
```

Здесь `X` — переменная типа `tObj`, а `Name` — глобальная переменная, экспортированная сканером.

Далее необходимо учесть, что обозначение процедуры может быть составным — состоять из имени модуля и имени процедуры, разделенных точкой. Поэтому первое встретившееся имя может оказаться именем модуля. В общем случае составное обозначение может быть и у переменной, константы, типа, если они импортированы из другого модуля. Однако стандартные модули `In` и `Out` языка «О» экспортируют только процедуры.

Если найденное в таблице имя принадлежит модулю, то проверяется наличие точки, имени импортируемого объекта и выполняется новый поиск. При этом в качестве ключа процедуре `Find` передается (после проверки длины) составное обозначение, состоящее из имени модуля, точки и имени искомого объекта, например, `'In.Int'`. Имен-

но таким образом обозначения процедур из стандартных модулей были занесены в таблицу имен при ее инициализации.

```
      if X^.Cat = catModule then begin
         NextLex;
         Check(lexDot, '"."');
         if    (Lex = lexName) and
            (Length(Name) + Length(X^.Name) < NameLen)
         then
            Find(X^.Name + '.' + Name, X)
         else
            Expected('имя из модуля ' + X^.Name);
      end;
```

Результат поиска составного имени (ссылка на найденный в таблице имен объект) вновь помещается в переменную X. Если же имя модуля не встретилось, то последующей части программы будет передана прежняя ссылка X, а эта последующая часть даже «не узнает», было обозначение составным или нет.

Дальнейший анализ не составляет труда. Окончательный текст распознавателя операторов приведен в листинге 3.37.

Листинг 3.37. Распознавание операторов

```
procedure Statement;
var
   X : tObj;
begin
   if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catModule then begin
         NextLex;
         Check(lexDot, '"."');
         if (Lex = lexName) and
            (Length(X^.Name) + Length(Name) < NameLen)
         then
            Find(X^.Name + '.' + Name, X)
         else
            Expected('имя из модуля ' + X^.Name);
      end;
      if X^.Cat = catVar then
         AssStatement            {Присваивание}
      else if (X^.Cat = catStProc) and
              (X^.Typ = typVoid)
      then
         CallStatement(X^.Val) {Вызов процедуры}
```

```
        else
           Expected(
              'обозначение переменной или процедуры'
           );
        end
     else if Lex = lexIF then
        IfStatement
     else if Lex = lexWHILE then
        WhileStatement
  end;
```

Полагаю, принцип контекстных проверок, выполняемых с помощью таблицы имен, понятен. Рассмотренные выше трансляция описаний, анализ выражений и распознавание видов операторов дают достаточно примеров. Мы не будем сейчас подробно обсуждать контекстный анализ конкретных операторов языка «О». В дальнейшем эти части программы все равно пришлось бы переписывать еще раз для внедрения действий по генерации кода. Чтобы не загромождать изложение, рассмотрим контекстный анализ отдельных операторов позже, вместе с генерацией кода для них.

Генерация кода

Интересное дело: существенная часть компилятора уже написана, а еще не было сказано ни слова о процессоре, в код которого должна транслироваться программа на языке «О». И это должно радовать, поскольку свидетельствует об универсальности использованных алгоритмов и хорошем проектировании программы, отдельные части которой максимально независимы.

Виртуальная машина

Мы не будем здесь рассматривать генерацию кода для какого-либо реального процессора или семейства процессоров. Во-первых, потому, что различных процессоров существует множество, и нам пришлось бы выбирать какой-то из них. Во-вторых, генерация кода для реального процессора была бы сопряжена с необходимостью учета множества технических деталей, не так уж важных в принципиальном плане. Немало места пришлось бы уделить рассмотрению самой системы команд процессора.

Поступим по-другому. Сконструируем собственный процессор, обладающий простой и удобной системой команд. Используя его, мы сможем рассмотреть основные принципы генерации кода, не отвлека-

ясь на частности. Речь, конечно, не идет об изготовлении прибора в металле и кремнии. Вместо этого используем программу, имитирующую работу процессора.

В выбранном подходе можно увидеть еще ряд достоинств. Программа, оттранслированная в код не существующего реально, а моделируемого программно (виртуального) процессора сможет быть выполнена в любой системе, где будет способен работать имитатор-интерпретатор. А поскольку написан он будет на Паскале, это означает, что программы на языке «О» можно будет выполнить везде, где есть подходящий компилятор языка Паскаль. Наконец, выбранный нами подход похож на технологии, использованные при реализации языков Ява и Си#, а еще раньше — Паскаля. Это позволяет познакомиться с принципами, лежащими в основе этих технологий.

Несуществующий абстрактный компьютер, работа которого реализуется на реальной машине с помощью программных средств, называют *виртуальной машиной*. Примером является Java Virtual Machine (JVM) — виртуальная Ява-машина, представляющая собой модель стекового процессора, в код которого транслируются программы на языке Ява. Дадим название и нашему компьютеру. Называться он будет ОВМ — виртуальная О-машина.

Исторически использование программного моделирования гипотетического компьютера связано с одной из первых реализаций языка Паскаль в начале 1970-х годов. Виртуальный код был назван тогда П-кодом. Позднее такая же техника использовалась при реализации Visual Basic. В системе Microsoft.NET код виртуальной машины носит название «промежуточный язык» (intermediate language, сокращенно — IL).

Архитектура виртуальной машины

Как и любой компьютер, ОВМ (рис. 3.6) будет содержать процессор и память. Процессор способен выполнять определенный набор команд, а также содержит ряд регистров — специальных ячеек памяти, используемых командами. В памяти хранятся программа и данные.

Память

Поскольку в языке «О» используются лишь данные целого типа, память ОВМ будет состоять из некоторого количества слов-ячеек, каждая из которых может хранить одно целое число. Каждое слово

имеет уникальный адрес, который выражается целым числом. В программе, моделирующей работу виртуальной машины, это будет выглядеть следующим образом:

```
const
   MemSize = 8*1024;

var
   M: array [0..MemSize-1] of integer;
```

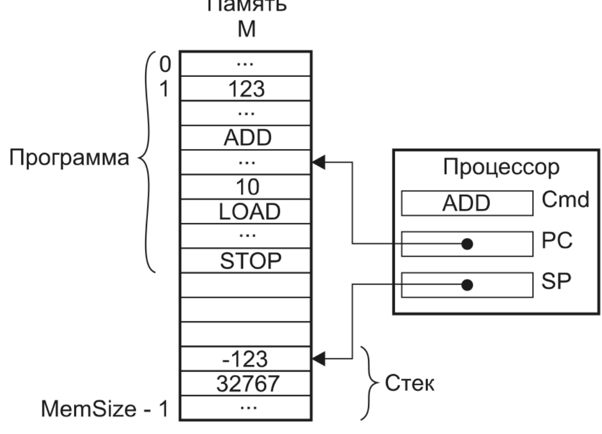


Рис. 3.6. Архитектура ОВМ

Константа `MemSize` определяет размер памяти в словах. Для примера ее значение взято равным 8К слов. Такой памяти вполне достаточно для размещения небольших демонстрационных программ, написанных на языке «О», а также их данных. Поскольку массивы в языке не предусмотрены, объем данных не может быть слишком велик.

Массив `M` (от memory — память) — это память виртуальной машины. Номера (адреса) его элементов начинаются с нуля.

Разрядность слова виртуальной машины зависит от разрядности типа `integer` в том компиляторе Паскаля, с помощью которого будет транслироваться интерпретатор виртуальной машины. Собственно, реальных вариантов два: при 16-разрядном представлении (Turbo Pascal, Free Pascal) ОВМ будет 16-разрядной; если используется 32-разрядный тип `integer` (Delphi, Pascal ABC и др.), ОВМ тоже будет 32-разрядной.

Разрядность слова определяет не только диапазон целых чисел, которые можно обрабатывать, но и размер адресного пространства машины. Поскольку адреса ячеек памяти сами будут храниться в таких же ячейках памяти, максимально возможное значение адреса 16-разрядной машины составит 32 767; 32-разрядной — 2 147 483 647. Таким образом, максимальный объем памяти 16-разрядной ОВМ мог бы составить 32К слов (64 Кбайт); 32-разрядной — 2 гигаслова (8 Гбайт). Реально, ни то, ни другое недостижимо: 16-разрядная реализация Паскаля не позволит создать массив объемом 64 Кбайт, реальный 32-разрядный компьютер, для которого создают код 32-разрядные компиляторы, не может иметь больше 4 Гбайт памяти.

Процессор

Одной из форм представления программы при трансляции является обратная польская запись (ПОЛИЗ). Преобразование программы в ПОЛИЗ (генерация кода), и ее последующее исполнение (интерпретация) выполняются простым и естественным способом с использованием стека. В связи с этим, в роли виртуального процессора используем стековую машину с безадресной (нуль-адресной[75]) системой команд. Программа для такой машины представляет собой последовательность операндов и операций, соответствующих обратной польской записи программы. Встретившиеся в программе операнды заносятся в стек, операции выполняются над верхними элементами стека, результат операции заменяет собой операнды на вершине стека.

Каждый операнд (константа, адрес) и код каждой операции в программе для ОВМ будет занимать одно слово памяти. Чтобы различать операнды и операции, предусмотрим кодирование операций отрицательными целыми числами. Тогда для операндов остаются зарезервированы неотрицательные целые значения. Получается, что отрицательный операнд не может быть непосредственно указан в программе. Это, однако, не должно создать серьезных проблем, поскольку отрицательные числа встречаются в реальных программах гораздо

[75] В том смысле, что сама команда не содержит адресов своих операндов. Операнды всегда в стеке. В противоположность этому возможны одноадресные, двухадресные, трехадресные и даже четырехадресные команды и системы команд, а также системы команд с переменной адресностью.

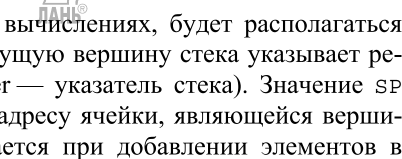
реже неотрицательных. К тому же, числовые литералы в языке «О» рассматриваются как числа без знака. Поэтому, например, в выражении `x-1` операндами являются `x` и 1, а минус — это знак двуместной операции. Чтобы было возможно выполнение действий, подобных `x := -1`, в системе команд ОВМ будет предусмотрена операция «перемена знака» — унарный минус.

Запрет отрицательных констант в машинном коде ОВМ не означает отказ от отрицательных чисел вообще, то есть запрет на получение в стеке отрицательных величин в ходе вычислений и хранение отрицательных значений в ячейках данных, отведенных для переменных.

Программный счетчик

Команды программы располагаются в памяти ОВМ (массиве `M`). Процессор выполняет команды одну за другой. При отсутствии переходов команды выполняются в порядке их расположения в памяти. Каждая команда — это либо операнд, либо операция. Условимся, что первой выполняется команда, записанная в ячейке с адресом 0.

Адрес очередной команды, подлежащей исполнению, записан в регистре процессора, обозначаемом `PC` (от program counter — программный счетчик). Начальное значение `PC` в соответствии с только что принятым соглашением равно 0.



Стек и указатель стека

Стек, используемый ОВМ при вычислениях, будет располагаться в старших адресах памяти. На текущую вершину стека указывает регистр процессора `SP` (stack pointer — указатель стека). Значение `SP` в каждый момент времени равно адресу ячейки, являющейся вершиной стека. Значение `SP` уменьшается при добавлении элементов в стек, и увеличивается при их извлечении — стек растет в сторону меньших адресов. Перед выполнением программы стек пуст, а значение `SP` равно `MemSize`.

Система команд

Как уже говорилось, программа для ОВМ представляет собой записанную в память (массив `M`) последовательность операндов и операций. Каждый операнд и каждая операция занимают одно слово. При выполнении программы операнды заносятся в стек, операции выполняются над данными с вершины стека. Операнды можно рассматри-

вать как операции (с кодами от 0 до `maxint`). Действие такой операции состоит в том, что в стек заносится ее код. Команды перед выполнением считываются в регистр команд процессора, обозначаемый `Cmd`.

Команды ОВМ перечислены в таблице 3.3. Для пояснения их действия в графе «Стек» приводятся состояния стека до и после выполнения команды. Вершина стека считается расположенной справа. Например, запись `x, y → x+y` означает, что до выполнения команды (`ADD`) на стеке[76] располагались числа `x` и `y`, причем, `y` — на вершине стека, а `x` — под вершиной, а после выполнения — их сумма. Естественно, что кроме `x` и `y` в глубине стека перед выполнением команды `ADD` могли размещаться и другие данные, но поскольку они не участвуют в ее выполнении, то и не упоминаются.

Таблица 3.3. Система команд ОВМ

Код	Обозначение	Название	Стек	Действие
c>=0	Нет	Константа	→ c	
-1	STOP	Останов	Не меняется	
Арифметические операции				
-2	ADD	Сложение	x, y → x+y	
-3	SUB	Вычитание	x, y → x-y	
-4	MUL	Умножение	x, y → x*y	
-5	DIV	Деление	x, y → x **DIV** y	
-6	MOD	Остаток	x, y → x **MOD** y	
-7	NEG	Изменение знака	x → -x	
Операции с памятью				
-8	LOAD	Загрузка	A → M[A]	
-9	SAVE	Сохранение	A, x →	M[A] := x
Операции со стеком				
-10	DUP	Дублирование	x → x, x	

[76] Выражение «на стеке» — программистский жаргон. По правилам русского языка следовало, пожалуй, сказать «в стеке». Но было бы не очевидно, что речь идет о верхних элементах. «На стеке» — сокращенный вариант оборота «на вершине стека».

Продолжение табл. 3.3

Код	Обозначение	Название	Стек	Действие
-11	DROP	Сброс	x →	
-12	SWAP	Обмен	x, y → y, x	
-13	OVER	Наверх	x, y → x, y, x	
Команды перехода				
-14	GOTO	Безусловный переход	A →	PC := A
-15	IFEQ	Переход, если равно	x, y, A →	**if** x=y **then** PC := A
-16	IFNE	Переход, если не равно	x, y, A →	**if** x<>y **then** PC := A
-17	IFLE	Переход, если меньше или равно	x, y, A →	**if** x<=y **then** PC := A
-18	IFLT	Переход, если меньше	x, y, A →	**if** x<y **then** PC := A
-19	IFGE	Переход, если больше или равно	x, y, A →	**if** x>=y **then** PC := A
-20	IFGT	Переход, если больше	x, y, A →	**if** x>y **then** PC := A
Операции ввода и вывода				
-21	IN	Ввод	→ введенное число	SP := SP-1; Write('?'); Readln(M[SP])
-22	OUT	Вывод	x, w →	Write(x: w)
-23	OUTLN	Перевод строки	Не меняется	WriteLn

Такой способ обозначений заимствован из описаний языка Форт, да и вообще, система команд ОВМ устроена по тому же принципу, что и Форт.

Краткие пояснения по поводу некоторых команд. Как уже говорилось, все арифметические команды берут свои операнды с вершины стека. Двуместные операции (ADD, SUB, MUL, DIV, MOD) заменяют два операнда, взятые с вершины стека, результатом операции. При этом

число элементов в стеке уменьшается на единицу. Операция `NEG` меняет знак элемента, находящегося на вершине стека.

Команда `LOAD` загружает на вершину стека значение, хранящееся в памяти по указанному адресу `A`. Её действие сводится к замене на стеке адреса `A` значением `M[A]`. Такое действие называется «разыменованием». Команда `SAVE` сохраняет взятое со стека значение в ячейке памяти с указанным адресом. Перед выполнением `SAVE` на вершине стека должно быть сохраняемое значение, под вершиной — адрес.

Команды, оперирующие элементами на вершине стека, имеют тот же смысл и обозначения, что и в языке Форт. `DUP` дублирует элемент, находящийся на вершине стека, `DROP` уничтожает верхний элемент, `SWAP` обменивает два верхних элемента стека, `OVER` дублирует на вершине стека элемент, находившийся под вершиной.

Команда `GOTO` выполняет переход по адресу, находящемуся на вершине стека. Реализация перехода сводится к присваиванию значения с вершины стека программному счетчику `PC`. После этого следующей выполняемой командой будет команда, находящаяся в памяти по адресу, занесенному в `PC`.

Команды условных переходов требуют трех операндов на стеке. Сравниваемые значения `x` и `y` находятся под вершиной стека, адрес перехода `A` — на вершине. Происходит переход по адресу `A`, если выполняется заданное отношение для `x` и `y`, в противном случае выполняется следующая команда.

Команды ввода-вывода соответствуют имеющимся в языке «О» возможностям. Команда `IN` печатает знак «?», запрашивает вводимое число и заносит его на вершину стека. `OUT` выводит целое значение `x`, находящееся под вершиной стека, используя `w` позиций. `OUTLN` выполняет перевод строки.

Вполне можно представить себе аппаратный компьютер, имеющий такую систему команд. Разве что ввод и вывод организованы в ОВМ нетрадиционно. Конкретные операции ввода и вывода, как правило, не входят в систему команд реальных процессоров. Связь с внешними устройствами осуществляется через специальные порты ввода-вывода, либо для обмена с внешними устройствами резервируется определенная часть адресного пространства. В последнем случае обмен сводится к записи и считыванию данных по определенным адресам памяти. И уж совсем нетипично, что в набор команд ОВМ входят

такие действия как печать целого и перевод строки. В реальных системах речь могла бы идти о выводе отдельного символа, а вывод числа и перевод строки были бы реализованы программно.

Еще одна особенность, отличающая систему команд ОВМ от команд реальных процессоров, состоит в том, что для хранения 24 различных кодов операций используется целое машинное слово длиной 16 или даже 32 бита, в то время как было бы достаточно всего 5 бит.

Программирование в коде виртуальной машины

Чтобы освоить систему команд ОВМ и понять принципы программирования для этой машины, рассмотрим примеры. Возьмем задачу нахождения наибольшего общего делителя (НОД) двух натуральных чисел и запрограммируем ее вначале на языке «О», используя для решения алгоритм Евклида: пока числа не сравняются, уменьшать большее из них на величину меньшего (листинг 3.38).

Листинг 3.38. Нахождение НОД по алгоритму Евклида

```
(* Наибольший общий делитель *)
MODULE Euqlid;
IMPORT In, Out;
VAR
   X, Y : INTEGER;
BEGIN
   In.Open;
   In.Int(X);
   In.Int(Y);
   WHILE X # Y DO
      IF X > Y THEN
         X := X - Y
      ELSE
         Y := Y - X
      END;
   END;
   Out.Int(X, 0);
   Out.Ln;
END Euqlid.
```

Программирует компилятор

Представим, какой машинный код должен создать компилятор для такой программы на языке «О». Запишем этот код, имея в виду следующее. Компилятор назначает каждой переменной свою ячейку памяти. Для получения значения переменной генерируются команды,

загружающие это значение из памяти. Наш компилятор не выполняет никакой оптимизации, программируя «в лоб».

Вообще-то, если речь идет о машинном коде, следовало бы записать программу как последовательность чисел. Но читать ее в таком виде трудно. Используем мнемонические коды команд вместо числовых. Применим и некоторые другие обозначения. В языках машинного уровня (ассемблерах) часто используют точку с запятой для обозначения комментариев. Часть строки, следующая за точкой с запятой — комментарий. Строки программы на «О», породившие соответствующий машинный код, будем записывать в форме комментария. В той же строке, что и команда, будем отражать состояние стека после выполнения этой команды (вершина стека справа). Адреса команд отделяются круглой скобкой.

Распределим память под переменные X и Y. Поскольку код программы наверняка получится короче 100 команд, будем хранить значение X в ячейке 100, а значение Y — в ячейке 101. Константы 100 и 101 в листинге 3.39 означают адрес X и адрес Y соответственно.

Листинг 3.39. Машинный код программы Euqlid

```
; Наибольший общий делитель
; MODULE Euqlid;
; IMPORT In, Out;
; VAR
; X, Y : INTEGER;
; BEGIN
; In.Int(X);

   0) 100    ; 100
   1) IN     ; 100, X
   2) SAVE

; In.Int(Y);

   3) 101    ; 101
   4) IN     ; 101, Y
   5) SAVE

; WHILE X # Y DO

   6)  100   ; 100
   7)  LOAD  ; X
   8)  101   ; X, 101
```

```
   9)  LOAD  ; X, Y
   10) 36    ; X, Y, 36
   11) IFEQ

; IF X > Y THEN

   12) 100   ; 100
   13) LOAD  ; X
   14) 101   ; 101
   15) LOAD  ; Y
   16) 27    ; X, Y, 27
   17) IFLE

; X := X – Y

   18) 100   ; 100
   19) 100   ; 100, 100
   20) LOAD  ; 100, X
   21) 101   ; 100, X, 101
   22) LOAD  ; 100, X, Y
   23) SUB   ; 100, X-Y
   24) SAVE

; ELSE

   25) 34
   26) GOTO

; Y := Y – X

   27) 101   ; 101
   28) 101   ; 101, 101
   29) LOAD  ; 101, Y
   30) 100   ; 101, Y, 100
   31) LOAD  ; 101, Y, X
   32) SUB   ; 101, Y-X
   33) SAVE

; END;
; END;

   34) 6
   35) GOTO
```

ЛАНЬ®
ЛАНЬ®

```
; Out.Int(X, 0);

   36) 100   ; 100
   37) LOAD  ; X
   38) 0     ; X, 0
   39) OUT

; Out.Ln;

   40) OUTLN

; END Euqlid.

   41) STOP
```

Программа заняла 42 машинных слова. Теперь, зная размер кода, можно было бы изменить адреса X и Y, предусмотрев размещение этих переменных сразу за кодом программы. Переменной X можно назначить адрес 42, переменной Y — 43. Компилятор, по-видимому, должен будет действовать аналогично, ведь он не может выдвигать гипотезы о будущем размере кода и назначать адреса еще до начала компиляции «с запасом», как поступили мы. Переписывать программу, заменяя константы 100 и 101 на 42 и 43, не будем. Но будем иметь в виду, что при дальнейшей разработке компилятора задачу назначения адресов придется решать.

Программируем вручную

Программа, представленная в листинге 3.39, далеко не оптимальна. Оперируя всего двумя величинами X и Y, она постоянно занята загрузкой и сохранением их значений. Но зачем сохранять значения в памяти, если они тут же потребуются в следующем цикле? Можно в ходе выполнения удерживать X и Y на стеке. Действуя по такому принципу, напишем новый вариант программы (листинг 3.40).

Листинг 3.40. Нахождение НОД(X, Y). Программирование вручную

```
0)  IN    ; X
1)  IN    ; X, Y
2)  OVER  ; X, Y, X
3)  OVER  ; X, Y, X, Y
4)  15    ; X, Y, X, Y, 15
5)  IFEQ  ; X, Y    На выход, если X=Y
6)  OVER  ; X, Y, X
```

```
 7)  OVER  ; X, Y, X, Y
 8)  11    ; X, Y, X, Y, 11
 9)  IFLT  ; X, Y    В обход SWAP, если X>Y
10)  SWAP  ; Y, X    На вершине большее
11)  OVER  ; Min(X, Y), Max(X, Y), Min(X, Y)
12)  SUB   ; Новое X, Новое Y
13)  2     ; X, Y, 2
14)  GOTO  ; X, Y    На начало цикла
15)  DROP  ; X       Одно значение было лишним
16)  0     ; X, 0
17)  OUT
18)  OUTLN
19)  STOP
```

Достаточно трудно представить, что компилятор может породить такой код. Приведенная программа основана на неочевидных манипуляциях с вершиной стека. Такая манера характерна для ручного программирования на языке Форт. Сравнение листингов 3.39 и 3.40 наглядно демонстрирует причины меньшей эффективности программ, полученных трансляцией с языка высокого уровня[77] в сравнении с написанными вручную. Код, полученный вручную, оказался вдвое короче, и работать будет быстрее, поскольку в цикле написанной вручную программы выполняется 12 или 13 команд, а в откомпилированной программе — 21 или 23.

Для нахождения наибольшего общего делителя (НОД) двух натуральных чисел может быть написана еще более компактная программа[78] (листинг 3.41). Вместо вычитаний она использует вычисление остатка от деления.

Листинг 3.41. Нахождение НОД с вычислением остатка

```
0)  IN    ; X
1)  IN    ; X, Y
2)  SWAP  ; Y, X
3)  OVER  ; Y, X, Y
4)  MOD   ; Y, X mod Y
5)  DUP   ; Y, X mod Y, X mod Y
6)  0     ; Y, X mod Y, X mod Y, 0
7)  2
8)  IFNE
9)  DROP
```

[77] С помощью простого неоптимизирующего компилятора.

[78] Автор Ф. Меньшиков.

```
10) 0
11) OUT
12) OUTLN
13) STOP
```

Реализация виртуальной машины

Спроектировав архитектуру виртуальной машины и даже поупражнявшись в программировании ОВМ, займемся её воплощением. Роль виртуальной машины будет исполнять ее программная модель — интерпретатор. Ресурсы, предоставляемые виртуальной машиной, будут сосредоточены в программном модуле `OVM`. В его интерфейсной секции (листинг 3.42) определяются константа `MemSize`, задающая размер памяти, константы, обозначающие коды операций, и массив `M` — память виртуальной машины.

Листинг 3.42. Интерфейс модуля виртуальной машины

```
unit OVM;
{Виртуальная машина}

interface

const
   MemSize = 8*1024;

   cmStop   = -1;

   cmAdd    = -2;
   cmSub    = -3;
   cmMult   = -4;
   cmDiv    = -5;
   cmMod    = -6;
   cmNeg    = -7;

   cmLoad   = -8;
   cmSave   = -9;

   cmDup    = -10;
   cmDrop   = -11;
   cmSwap   = -12;
   cmOver   = -13;

   cmGOTO   = -14;
   cmIfEQ   = -15;
```

```
   cmIfNE   = -16;
   cmIfLE   = -17;
   cmIfLT   = -18;
   cmIfGE   = -19;
   cmIfGT   = -20;

   cmIn     = -21;
   cmOut    = -22;
   cmOutLn  = -23;

var
   M: array [0..MemSize-1] of integer;

procedure Run;
```

Процедура `Run` выполняет программу, записанную в память виртуальной машины, начиная с команды, находящейся в `M[0]`. Эта процедура реализует работу процессора ОВМ.

Использование модуля `OVM` предполагается строить по следующей схеме. Генератор кода записывает команды программы прямо в память виртуальной машины (массив `M`). Массив памяти и коды операций определены в интерфейсе модуля OVM, чтобы быть доступными другим модулям компилятора. После того как код сгенерирован и записан в память ОВМ, программа может быть выполнена вызовом процедуры `Run`. Поместим этот вызов в главную программу нашего компилятора (листинг 3.43).

Листинг 3.43. Главная программа транслятора языка «О»

```
program O;
{Компилятор языка O}

uses
   OText, OScan, OPars, OVM, OGen;

procedure Init;
begin
   ResetText;
   InitScan;
   InitGen;
end;

procedure Done;
begin
   CloseText;
```

```
end;

begin
   WriteLn('Компилятор языка O');
   Init;     {Инициализация}
   Compile;  {Компиляция}
   Run;      {Выполнение}
   Done;     {Завершение}
end.
```

После такого решения транслятор приобрел черты интерпретатора, поскольку отвечает теперь и за выполнение программы. Если иметь в виду учебный характер проекта, такое сочетание компиляции и интерпретации безусловно полезно — позволяет понять реализацию как одного, так и другого. В реальных системах интерпретаторы обычно строятся по схожей схеме — интерпретируется не исходная программа, а ее промежуточное представление. Это позволяет достичь большей эффективности.

Приведенный в листинге 3.43 текст главной программы транслятора языка «О» является окончательным. Обратите внимание, что в предложении **uses** упомянут модуль генератора кода `OGen`, а в процедуре `Init` вызывается инициализация генератора кода (`InitGen`).

Реализация процедуры `Run` достаточно проста (листинг 3.44). Она строится по спецификации ОВМ. Регистры виртуальной машины `PC` и `SP` и `Cmd` превращаются в локальные переменные процедуры `Run`. Кроме этого используется локальная переменная `Buf`, необходимая для реализации команды `SWAP`. В начале работы регистры процессора приводятся в исходное состояние: `SP = MemSize` (стек пуст); `PC = 0` (выполнение начинается с команды, расположенной по адресу 0).

Обратите внимание, что сразу после считывания очередной команды в регистр команд `Cmd` и еще до начала выполнения команды программный счетчик `PC` увеличивается на единицу. Это означает, что при выполнении данной команды `PC` указывает на следующую по порядку команду.

Напомню, что стек ОВМ растет в сторону меньших адресов, поэтому при добавлении элементов в стек регистр `SP` уменьшается, при удалении — увеличивается. `M[SP]` означает текущую вершину стека, `M[SP+1]` — элемент, находящийся под вершиной, `M[SP+2]` — второй от вершины элемент в глубине стека. При реализации двуместных арифметических операций, уменьшающих число элементов в стеке на

единицу, вначале изменяется указатель стека, после чего `M[SP]` — это «новая» вершина стека, а `M[SP-1]` — «старая».

Листинг 3.44. Процессор виртуальной машины

```
procedure Run;
var
   PC    : integer;
   SP    : integer;
   Cmd   : integer;
   Buf   : integer;
begin
   PC := 0;
   SP := MemSize;
   Cmd := M[PC];
   while Cmd <> cmStop do begin
      PC := PC + 1;
      if Cmd >= 0 then begin
         SP := SP - 1;
         M[SP] := Cmd;
         end
      else
         case Cmd of
         cmAdd:
            begin
               SP := SP + 1;
               M[SP] := M[SP] + M[SP-1];
            end;
         cmSub:
            begin
               SP := SP + 1;
               M[SP] := M[SP] - M[SP-1];
            end;
         cmMult:
            begin
               SP := SP + 1;
               M[SP] := M[SP]*M[SP-1];
            end;
         cmDiv:
            begin
               SP := SP + 1;
               M[SP] := M[SP] div M[SP-1];
            end;
         cmMod:
            begin
               SP := SP + 1;
```

```
            M[SP] := M[SP] mod M[SP-1];
         end;
      cmNeg:
         M[SP] := -M[SP];
      cmLoad:
         M[SP] := M[M[SP]];
      cmSave:
         begin
            M[M[SP+1]] := M[SP];
            SP := SP + 2;
         end;
      cmDup:
         begin
            SP := SP - 1;
            M[SP] := M[SP+1];
         end;
      cmDrop:
         SP := SP + 1;
      cmSwap:
         begin
            Buf := M[SP];
            M[SP] := M[SP+1];
            M[SP+1] := Buf;
         end;
      cmOver:
         begin
            SP := SP - 1;
            M[SP] := M[SP+2];
         end;
      cmGOTO:
         begin
            PC := M[SP];
            SP := SP + 1;
         end;
      cmIfEQ:
         begin
            if M[SP+2] = M[SP+1] then
               PC := M[SP];
            SP := SP + 3;
         end;
      cmIfNE:
         begin
            if M[SP+2] <> M[SP+1] then
               PC := M[SP];
```

```
      SP := SP + 3;
   end;
cmIfLE:
   begin
      if M[SP+2] <= M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfLT:
   begin
      if M[SP+2] < M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfGE:
   begin
      if M[SP+2] >= M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIfGT:
   begin
      if M[SP+2] > M[SP+1] then
         PC := M[SP];
      SP := SP + 3;
   end;
cmIn:
   begin
      SP := SP - 1;
      Write('?');
      Readln( M[SP] );
   end;
cmOut:
   begin
      Write(M[SP+1]:M[SP]);
      SP := SP + 2;
   end;
cmOutLn:
   WriteLn;
else begin
   WriteLn('Недопустимый код операции');
   M[PC] := cmStop;
end;
end;
```

```
      Cmd := M[PC];
   end;
   WriteLn;
   if SP < MemSize then
      WriteLn('Код возврата ', M[SP]);
   Write('Нажмите ВВОД');
   ReadLn;
end;
```

Если после выполнения команды STOP стек не будет пустым, то находящееся на его вершине число воспринимается как код возврата, переданный программой в «окружающую среду» и свидетельствующий о характере завершения программы. Значение 0 обычно интерпретируется как нормальное завершение, а отличные от нуля коды возврата соответствуют разным вариантам аварийного завершения.

Генератор кода

Было бы неправильно разрешать анализатору (модуль OPars) оперировать непосредственно с памятью виртуальной машины, записывать туда команды формируемой программы. Надо, насколько возможно, разделить анализирующую и генерирующую части компилятора. Это улучшит его структуру и придаст гибкость. Порождение машинных команд с помощью небольшого набора процедур, сосредоточенных в отдельном модуле, позволит лучше контролировать процесс генерации и при необходимости вносить в него изменения.

За непосредственную генерацию кода будет отвечать модуль OGen. Только процедурам этого модуля будет разрешено обращаться к памяти виртуальной машины для записи туда кодов машинных команд. Но решения о том, какие именно команды должны быть порождены, будут приниматься по ходу анализа входной программы, и вызовы процедур генерации будут размещены в недрах распознавателя.

В листинге 3.45 можно видеть первую версию модуля генератора кода. Основной его процедурой является Gen. Она записывает в очередную свободную ячейку памяти (массив M) команду, код которой передан при вызове (параметр Cmd). Адрес свободной ячейки хранится в глобальной и видимой из других модулей переменной PC. Это программный счетчик периода генерации. После записи в память очередной команды его значение увеличивается на единицу и снова начинает указывать на свободную ячейку. Перед началом генерации кода и всего процесса компиляции память виртуальной машины пу-

ста. Начальное значение PC, равное нулю, устанавливается процедурой InitGen, которая вызывается из главной программы компилятора.

Листинг 3.45. Предварительная версия генератора кода

```
unit OGen;
{Генератор кода}

interface

var
   PC : integer;

procedure InitGen;
procedure Gen(Cmd: integer);

{======================================================}

implementation

uses
   OVM;

procedure InitGen;
begin
   PC := 0;
end;

procedure Gen(Cmd: integer);
begin
   M[PC] := Cmd;
   PC := PC+1;
end;

end.
```

Не следует путать переменную PC, определенную в модуле OGen и программный счетчик виртуальной машины. Последний — это локальная переменная виртуальной машины и недоступна извне. Программный счетчик периода генерации будет использоваться в модуле распознавателя (но только для того, чтоб узнать его текущее значение).

Распределение памяти

Каждой переменной программы на языке «О» должна быть назначена ячейка памяти, где будет храниться значение этой переменной. Можно представить себе несколько вариантов размещения переменных в памяти ОВМ. Эти варианты показаны на рисунке 3.7.

Можно разместить переменные перед кодом программы в начале памяти (рис. 3.7*а*). Поскольку ОВМ при запуске начинает выполнение с команды по адресу 0, компилятор должен поместить в ячейки с адресами 0 и 1 команды, выполняющие переход на основную часть программы в обход участка памяти, отведенного под переменные. Адреса переменным компилятор может назначить в ходе трансляции описаний и хранить эти адреса в поле `Val` записей таблицы имен, относящихся к переменным. Первая переменная получает адрес 2, следующая — 3 и т. д.

Другой возможный вариант размещения переменных — в старших адресах памяти, на дне стека (рис. 3.7*б*). Первая переменная программы размещается в ячейке с адресом `MemSize-1`, вторая по адресу `MemSize-2` и т. д.

Назначение адресов может происходить уже в ходе трансляции описаний переменных. Первыми командами машинная программа должна выполнить резервирование места под переменные. Указатель стека должен быть продвинут вверх на столько ячеек, сколько переменных имеется в программе (значение `SP` должно быть уменьшено на число переменных программы). Резервирование заданного количества ячеек в стеке может быть выполнено серией команд, записывающих в стек константу (например, 0). Это можно сделать в цикле или индивидуально для каждой переменной. Запись определенной константы в отведенную переменной ячейку при резервировании памяти означало бы гарантированную инициализацию переменных. В тех языках, где такая инициализация предусмотрена, подобный механизм распределения был бы весьма подходящим. В языке «О» обязательная инициализация не предусмотрена.

На рисунке 3.7*в* показан вариант с таким же размещением переменных. Но резервирование места в стеке выполняется специальной командой `ENTER`, которой пока нет в системе команд ОВМ. Эта команда будет рассмотрена позже.

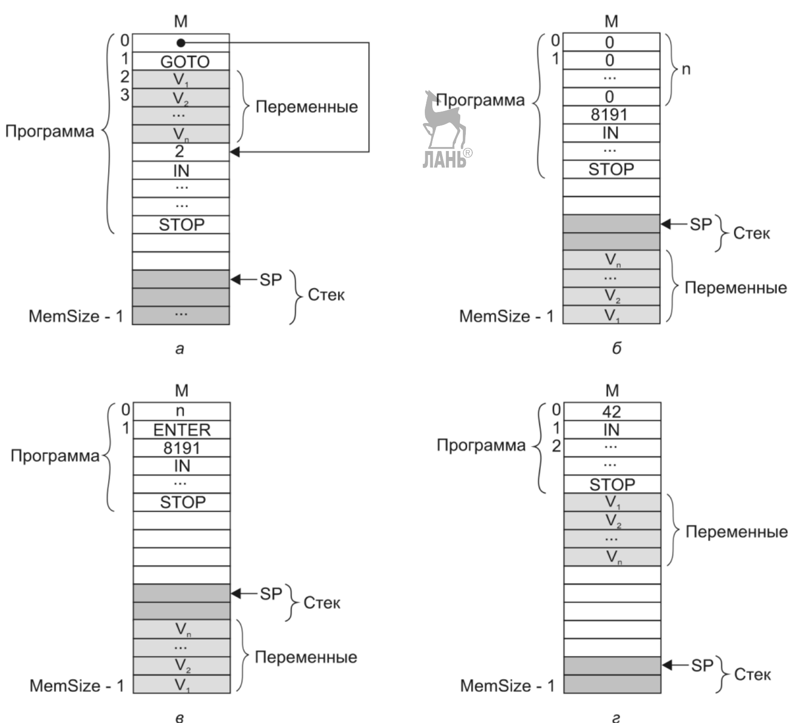


Рис. 3.7. Варианты распределения памяти под переменные

Одним из недостатков размещения переменных на дне стека является то, что код программы оказывается зависим от объема памяти виртуальной машины. Программа, откомпилированная для машины с памятью, например, 16К слов не сможет выполняться на машине с памятью в 8К слов, даже если этого объема памяти было бы достаточно. Дело в том, что в программе предназначенной для машины с памятью 16К слов, переменные размещены в ячейках с адресами 16 383, 16 384, 16 381, ..., которых просто нет у машины с 8К слов.

Наиболее естественным представляется размещение переменных сразу за кодом программы после команды `STOP` (рис. 3.7*г*). Именно такой вариант и будет использован в нашем компиляторе. Вопрос только в том, что объем кода неизвестен как во время трансляции описаний, так и при генерации команд, в которых должны использо-

ваться адреса переменных. Проблема, однако, решается. При этом не требуется формировать никаких дополнительных машинных команд для обхода области размещения переменных или резервирования ячеек. После того как мы приобретём опыт генерации кода, вернемся к этому вопросу и реализуем размещение переменных несложным, хоть и не тривиальным способом. При этом окажется, что задержка с назначением адресов переменным до завершения генерации кода не препятствует этой генерации.

Для полноты картины упомянем вариант, когда переменные размещаются после любой имеющейся в программе команды безусловного перехода. Это не требует генерации никаких дополнительных команд, но представляется ненужной экзотикой и не имеет каких-либо преимуществ. Вообще говоря, память под переменные даже не обязательно распределять единым массивом.

Под константы также можно было бы отводить отдельные ячейки памяти и при необходимости доступа к их значениям загружать константы на стек из этих ячеек. Но в нашем случае в этом нет нужды, поскольку константы могут быть просто встроены в код.

Генерация кода для выражений

Общий принцип генерации кода для выражений состоит в том, что каждая распознающая процедура, участвующая в трансляции — транслятор выражения, простого выражения, слагаемого, множителя — должна сформировать такой машинный код, который вычисляет значение соответствующего подвыражения, оставляя вычисленное значение на вершине стека. Или по-другому: каждый распознаватель формирует обратную польскую запись подвыражения, за которое этот распознаватель отвечает — ведь машинный код ОВМ — не что иное, как обратная польская запись программы.

Генерация кода для множителей

Множитель — это элементарное выражение, атомарный операнд: константа, переменная, вызов функции, выражение в скобках.

Код для константы, встретившейся в выражении, должен поместить значение этой константы на стек. Для неотрицательных констант достаточно сгенерировать команду, совпадающую со значением константы, для отрицательных предусматривается еще перемена знака.

Код, порождаемый при распознавании переменной в составе выражения, должен загружать значение переменной на стек. Для этого генерируется адрес переменной и команда LOAD.

Для вызова каждой стандартной функции код генерируется индивидуально. Формирование кода для выражения в скобках распознаватель множителя поручает распознавателю выражений.

В листинге 3.46 приведена процедура Factor (ее предыдущий вариант — в листинге 3.35) — анализатор множителя, в текст которого добавлены действия по генерации кода, которые выделены.

Листинг 3.46. Генерация кода для множителя

```
procedure Factor(var T : tType);
var
   X : tObj;
begin
   if Lex = lexName then begin
      Find(Name, X);
      if X^.Cat = catVar then begin
         GenAddr(X);     {Адрес переменной}
         Gen(cmLoad);
         T := X^.Typ;
         NextLex;
         end
      else if X^.Cat = catConst then begin
         GenConst(X^.Val);
         T := X^.Typ;
         NextLex;
         end
      else if (X^.Cat=catStProc) and (X^.Typ<>typVoid)
      then begin
         NextLex;
         Check(lexLPar, '"("');
         StFunc(X^.Val, T);
         Check(lexRPar, '")"');
         end
      else
         Expected(
         'переменная, константа или процедура-функции'
         );
      end
   else if Lex = lexNum then begin
      T := typInt;
      GenConst(Num);
```

```
      NextLex
      end
   else if Lex = lexLPar then begin
      NextLex;
      Expression(T);
      Check(lexRPar, '")"');
      end
   else
      Expected('имя, число или "("');
end;
```

Явным образом здесь генерируется лишь команда LOAD для загрузки на стек значения переменной (код команды задан константой cmLoad, определенной в модуле виртуальной машины, см. листинг 3.42). Для генерации кода для константы вызывается процедура GenConst, которая порождает одну или две команды в зависимости от знака константы. Отрицательное число не может непосредственно быть записано в код ОВМ, поскольку в этом случае его не отличить от кода операции. Напомню, что все коды операций ОВМ — отрицательные. Процедуру GenConst (листинг 3.47) размещаем в модуле OGen.

Листинг 3.47. Генерация кода для константы (модуль OGen)

```
procedure GenConst(C: integer);
begin
   Gen(abs(C));
   if C < 0 then
      Gen(cmNeg);
end;
```

Должно быть понятно, что процедура GenAddr(X), не может записать в память ОВМ адрес переменной, который будет этой переменной окончательно назначен. Код программы еще не сформирован полностью и невозможно узнать адрес ячейки, где могла бы разместиться та переменная, на запись об имени которой в таблице имен ссылается указатель X. В действительности GenAddr записывает в то место машинной программы, где должен быть адрес, некоторую служебную информацию, которая в дальнейшем позволит поместить сюда правильный адрес. До рассмотрения алгоритма назначения адресов и реализации GenAddr мы можем считать, что действие этой процедуры соответствует ее названию.

Трансляция вызовов стандартных функций

В языке «О» предусмотрены четыре стандартных функции: ABS, MAX, MIN и ODD. Это то подмножество стандартных функций Оберона, которое сохраняет смысл в языке, где есть переменные и константы только целого типа, а также простейшие логические выражения.

Обычно основанием для включения функции или процедуры непосредственно в язык является то, что она не может быть запрограммирована с помощью других средств этого языка, поскольку имеет нестандартный синтаксис фактических параметров, необычные правила, относящиеся к типам параметров и типу значения. Если иметь в виду Оберон, это, безусловно, относится к MAX и MIN, поскольку аргументом этих функций является тип, в то время как обычные процедуры и функции Оберона не могут иметь аргументов-типов. Функция ABS в Обероне также своеобразна: тип ее значения зависит от типа аргумента. Абсолютная величина вещественного числа будет иметь вещественный тип, целого — целый. Что касается ODD, то она могла быть запрограммирована обычными средствами и присутствует в Обероне, вероятно, по традиции, обеспечивая эффективное преобразование целого типа в логический.

Такой статус стандартных функций обуславливает и способ из реализации в компиляторе. Во-первых, для вызовов стандартных функций зачастую генерируются не команды вызова подпрограмм, а машинный код, вычисляющий значение функции прямо в месте вызова. Во-вторых, обработка списка фактических параметров стандартных функций и процедур выполняется индивидуально.

В нашем компиляторе обработку списка фактических параметров выполняет процедура StFunc.

```
procedure StFunc(F: integer; var T: tType);
```

Входным параметром StFunc является номер стандартной функции (stABS, stMAX, ...); выходным — тип ее результата. В момент вызова StFunc (см. листинги 3.35, 3.46) текущей лексемой является первая лексема списка фактических параметров.

Рассмотрим вначале, как должны транслироваться отдельные функции и их списки аргументов.

Функция ABS

ABS имеет единственный аргумент, который может быть выражением. Транслируем это выражение, вычисляя его тип и генерируя для него код:

```
Expression(T);
```

В языке «О» аргумент ABS может иметь только целый тип. Проверяем это:

```
if T <> typInt then
   Expected('выражение целого типа');
```

Транслятор выражений Expression генерирует команды, в результате исполнения которых значение фактического параметра ABS будет помещено на вершину стека. Теперь наша задача состоит в том, чтобы сформировать машинный код, который изменит знак числа, находящегося на вершине стека, если число отрицательно, и оставит число без изменения в противном случае. Напомню, что команды ОВМ, выполняющие проверки (команды условных переходов), удаляют свои операнды со стека. Поэтому перед проверкой необходимо продублировать с помощью DUP проверяемое значение. Код должен получаться такой:

```
A-5) DUP   ; X, X
A-4) 0     ; X, X, 0
A-3) A     ; X, X, 0, A
A-2) IFGE  ; X
A-1) NEG   ; -X
  A) …     ; теперь на стеке ABS(X)
```

Здесь X — значение аргумента функции; A — адрес команды, следующей за формируемым фрагментом, он же — адрес условного перехода при X>=0. В комментариях после точки с запятой, как обычно, показано состояние стека после выполнения каждой команды. Все приведенные команды, кроме той, что представляет собой адрес перехода A, могут быть легко сгенерированы вызовом процедуры Gen с соответствующим аргументом. Адрес перехода вычисляется добавлением к текущему значению счетчика команд (переменная PC модуля OGen) смещения ячейки, на которую выполняется переход, относительно ячейки, в которую записывается адрес этого перехода (ячейка с адресом A-3). Это смещение равно 3. Генерацию такого кода выполнит процедура GenAbs (листинг 3.48), которую поместим в модуль OGen.

Листинг 3.48. Генерация кода для ABS

```
procedure GenAbs;
begin
   Gen(cmDup);
   Gen(0);
   Gen(PC+3);
   Gen(cmIfGE);
   Gen(cmNeg);
end;
```

Вызовом GenAbs и заканчивается трансляция списка фактических параметров стандартной процедуры-функции ABS.

Вызов транслятора выражений Expression с последующей проверкой типа выражения будет несколько раз использоваться и в дальнейшем. Предусмотрим для этих целей специальные процедуры (листинг 3.49). Одна из них, IntExpression, будет вызываться, если надо транслировать выражение и проверить, что оно имеет целый тип, другая, BoolExpression, вызывается, если ожидается выражение логического типа.

Листинг 3.49. Трансляция и проверка типа выражений

```
procedure IntExpression;
var T : tType;
begin
   Expression(T);
   if T <> typInt then
      Expected('выражение целого типа');
end;

procedure BoolExpression;
var
   T : tType;
begin
   Expression(T);
   if T <> typBool then
      Expected('логическое выражение');
end;
```

Обе эти процедуры располагаются в секции реализации модуля OPars.

Процедура StFunc, включающая окончательный вариант фрагмента, отвечающего за трансляцию ABS, показана в листинге 3.50. Схема трансляции трех других функций та же самая: вызывается процедура,

распознающая аргумент, генерируется код самой функции, фиксируется тип функции.

Листинг 3.50. Трансляция стандартных функций

```
procedure StFunc(F: integer; var T: tType);
begin
   case F of
   spABS:
      begin
         IntExpression;
         GenAbs;
         T := typInt;
      end;
   spMAX:
      begin
         ParseType;
         Gen(MaxInt);
         T := typInt;
      end;
   spMIN:
      begin
         ParseType;
         GenMin;
         T := typInt;
      end;
   spODD:
      begin
         IntExpression;
         GenOdd;
         T := typBool;
      end;
   end;
end;
```

Функции MAX и MIN

Функция `MAX`, которая в языке «О» может быть использована только в форме `MAX(INTEGER)`, возвращает число, обозначаемое в Паскале `MaxInt`. Машинный код, порождаемый `MAX`, будет состоять из одной команды-константы, равной 32 767, если реализация 16-разрядная, и равной 2 147 483 647, если 32-разрядная. Кроме генерации команды фрагмент, отвечающий за `MAX`, проверяет аргумент вызовом `ParseType` и задает тип: `T := typInt`.

Несколько сложнее дело обстоит с трансляцией `MIN`. Виртуальная О-машина не предусматривает непосредственную загрузку отрицательной константы на стек. Формировать значение `MIN(INTEGER)` на стеке приходится в соответствии с формулой `–MAX(INTEGER) - 1`. Эту работу выполняет процедура `GenMin` (листинг 3.51) модуля `OGen`.

Листинг 3.51. Генерация кода для функции MIN

```
procedure GenMin;
begin
   Gen(MaxInt);
   Gen(cmNeg);
   Gen(1);
   Gen(cmSub);
end;
```

Функция ODD

Функция `ODD`, в отличие от `ABS`, `MAX` и `MIN`, имеет логический тип. Это, однако, не означает, что сгенерированный для нее код будет вычислять логическое значение, оставляя его на стеке. Логические выражения вообще и функция `ODD` в частности используются в языке «О» только в операторах **`IF`** и **`WHILE`**. И в том и в другом случае при истинности логического выражения должны выполняться операторы, машинный код которых следует сразу за условием (рис. 3.8). Если условие ложно, должен выполняться переход на участок машинного кода, расположенный дальше (переход вперед).

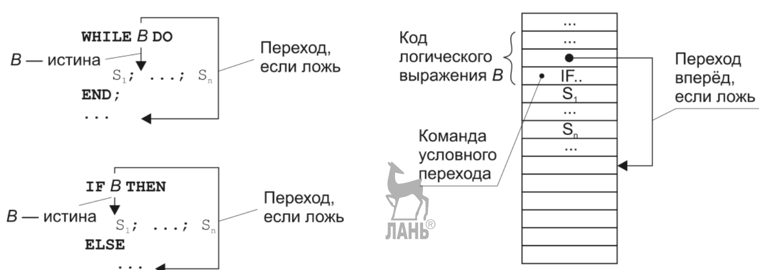


Рис. 3.8. Схема трансляции логических выражений

Таким образом, код логического выражения должен заканчиваться командой условного перехода, выполняемого, если выражение ложно. Это весьма универсальный подход, применимый в большинстве случаев и при трансляции логических выражений общего вида, вклю-

чающих операции «И», «ИЛИ», «НЕ». Можно также заметить, что и при трансляции оператора **REPEAT** – **UNTIL**, имеющегося в Обероне и других языках, логическое выражение также должно порождать «переход, если ложь». Что касается языка «О», то сформулированный принцип будет использован во всех случаях трансляции логических выражений.

Итак, код, который генерируется для ODD, должен вычислить остаток от деления находящегося на стеке числа на 2 и, если этот остаток равен 0 (число четно, значение ODD «ложь»), выполнить переход вперед. Однако, адрес этого перехода в момент генерации кода для ODD неизвестен. Его можно определить лишь после того, как будут сгенерированы команды для участка программы, выполняемого в случае истинности логического выражения. Если речь о цикле **WHILE**, то это следующее после слова **DO** тело цикла, а если логическое выражение используется в операторе **IF**, то это последовательность операторов, следующая за **THEN**.

Решение проблемы состоит в том, что вместо реального адреса перехода генерируется фиктивное значение (например, 0). Это позволяет продолжить генерацию, зарезервировав в машинном коде место, куда позднее будет помещен адрес перехода вперед. Местоположение ячейки, в которую занесен фиктивный адрес, запоминается, а после того, как становится известно место, куда должен быть выполнен переход, в эту ячейку записывается нужное значение.

Процедура, генерирующая код для ODD в соответствии с рассмотренной схемой, показана в листинге 3.52.

Листинг 3.52. Генерация кода для функции ODD

```
procedure GenOdd;
begin
   Gen(2);
   Gen(cmMod);
   Gen(0);
   Gen(0); {Фиктивный адрес перехода вперед}
   Gen(cmIfEQ);
end;
```

Запоминание местоположения ячейки, в которую записан фиктивный адрес перехода вперед, выполняется не внутри процедуры GenOdd, или процедур транслирующих другие варианты логических выражений, а в тех частях транслятора, которые отвечают за обработ-

ку операторов **IF** и **WHILE**. При трансляции этих операторов детальные сведения о коде, сформированном для логического выражения, следующего за **IF** и **WHILE** (а также **ELSIF**), уже будут недоступны. Однако остается известным, что машинный код логического выражения обязательно завершается командой условного перехода, а во второй от конца этого кода ячейке записан фиктивный адрес перехода вперед.

Невозможность определить адрес перехода вперед в момент формирования команды перехода — одна из причин того, что ранние компиляторы должны были выполнять несколько проходов. Не имея возможности удерживать в памяти сколько-нибудь значительные участки формируемого кода, многопроходный транслятор вначале запоминает ячейки, в которых должны быть записаны адреса переходов вперед, определяет эти адреса, и уже на другом проходе помещает их в соответствующие места формируемого кода.

Наличие памяти, позволяющей хранить весь формируемый машинный код (как в нашем случае) или хотя бы код одной процедуры, дает возможность выполнить формирование адресов переходов вперед однопроходному транслятору.

Генерация кода для слагаемых

Слагаемое (Term) — это один или несколько множителей, соединенных знаками операций типа умножения (*, **DIV**, **MOD**).

```
Слагаемое = Множитель {ОперУмн Множитель}.
```

Задача транслятора при преобразовании слагаемого в машинный код ОВМ состоит в том, чтобы породить команды, вычисляющие значение слагаемого и оставляющие это значение на вершине стека. При этом выражение вида

$$F_1 \otimes F_2,$$

где F_1, F_2 — множители, а $\otimes$ — операция типа умножения, транслируется в обратную польскую запись:

Машинный код для F_1
Машинный код для F_2
$\otimes$

Генерация кода для множителей выполняется вызовом процедуры `Factor`. Транслятору слагаемого (процедуре Term) остается лишь формировать команды, выполняющие операции умножения, деления

и получения остатка. Для этого знак операции запоминается в момент, когда соответствующая лексема является текущей, а после трансляции очередного множителя формируется машинная команда, соответствующая запомненному знаку.

Связанные с генерацией кода вставки, которые нужно сделать в процедуру `Term` из листинга 3.34, выполнявшую синтаксический и контекстный анализ слагаемого, в листинге 3.53 выделены.

Листинг 3.53. Транслятор слагаемого

```
procedure Term(var T: tType);
var
   Op : tLex;
begin
   Factor(T);
   if Lex in [lexMult, lexDIV, lexMOD] then begin
      if T <> typInt then
         Error
            ('Несоответствие операции типу операнда');
      repeat
         Op := Lex;
         NextLex;
         Factor(T);
         if T <> typInt then
            Expected('выражение целого типа');
         case Op of
         lexMult: Gen(cmMult);
         lexDIV:  Gen(cmDIV);
         lexMOD:  Gen(cmMOD);
         end;
      until not( Lex in [lexMult, lexDIV, lexMOD] );
   end;
end;
```

Генерация кода для простых выражений

Генерация кода для простых выражений выполняется аналогично трансляции множителей. Разница в том, что вместо умножения и деления речь идет о формировании машинных команд для операций сложения и вычитания, а также в том, что если перед первым слагаемым есть минус, то генерируется команда перемены знака.

Добавим в процедуру `SimpleExpr`, выполняющую синтаксический и контекстный анализ простого выражения (см. листинг 3.33), фрагменты, отвечающие за генерацию кода. В листинге 3.54 они выделены.

Листинг 3.54. Трансляция простого выражения

```
(* ["+"|"-"] Слагаемое {ОперСлож Слагаемое} *)
procedure SimpleExpr(var T : tType);
var
   Op : tLex;
begin
   if Lex in [lexPlus, lexMinus] then begin
      Op := Lex;
      NextLex;
      Term(T);
      if T <> typInt then
         Expected('выражение целого типа');
      if Op = lexMinus then
         Gen(cmNeg);
      end
   else
      Term(T);
   if Lex in [lexPlus, lexMinus] then begin
      if T <> typInt then
         Error
            ('Несоответствие операции типу операнда');
      repeat
         Op := Lex;
         NextLex;
         Term(T);
         if T <> typInt then
            Expected('выражение целого типа');
         case Op of
         lexPlus:  Gen(cmAdd);
         lexMinus: Gen(cmSub);
         end;
      until not( Lex in [lexPlus, lexMinus] );
   end;
end;
```

Генерация кода для выражений общего вида

Выражение языка «О» (и языка Оберон) — это одно простое выражение или два простых выражения, соединенных знаком операции отношения. В первом случае (одно простое выражение) никаких специальных действий по генерации кода не требуется, код формируется вызовом процедуры `SimpleExpr`.

Если же имеется знак отношения, то его следует запомнить в тот момент, когда этот знак прочитан, а после формирования кода для

правого операнда отношения (вызовом `SimpleExpr`) нужно сгенерировать условный переход вперед, выполняемый, если отношение ложно. Таким образом, выдерживается общее правило формирования кода для логических выражений: машинный код логического выражения заканчивается условным переходом вперед, выполняемым, если выражение ложно. Второй от конца этого кода командой является адрес перехода, который вначале заменяется фиктивным значением, а окончательно формируется распознавателем конструкции, в которой используется логическое выражение.

Вставки в анализатор выражений, отвечающие за генерацию кода, в листинге 3.55 отмечены.

Листинг 3.55. Генерация кода для выражений

```
 (* ПростоеВыраж [Отношение ПростоеВыраж] *)
procedure Expression(var T : tType);
var
  Op : tLex;
begin
  SimpleExpr(T);
  if Lex in [lexEQ, lexNE, lexGT, lexGE, lexLT, lexLE]
  then begin
    Op := Lex;
    if T <> typInt then
      Error('Несоответствие операции типу операнда');
    NextLex;
    SimpleExpr(T); {Правый операнд отношения}
    if T <> typInt then
      Expected('выражение целого типа');
    GenComp(Op);   {Генерация условного перехода}
    T := typBool;
  end;{иначе тип равен типу первого прост. выражения}
end;
```

Формирование условного перехода, замыкающего машинный код отношения (сравнения) выполняет процедура `GenComp` (от generate comparison — сформировать сравнение). Эту процедуру поместим в модуль `OGen`. В качестве параметра процедуре `GenComp` передается знак отношения, записанный в выражении. `GenComp` (листинг 3.56) формирует переход, используя машинную команду, соответствующую противоположному отношению. Так, например, если исходное сравнение имело вид `A<=B`, то будет сгенерирован «переход, если

больше» — команда IFGT, поскольку переход должен выполняться, если условие ложно.

Листинг 3.56. Генерация кода для сравнений

```
procedure GenComp(Op: tLex);
begin
   Gen(0); {Фиктивный адрес перехода вперед}
   case Op of
   lexEQ : Gen(cmIfNE);
   lexNE : Gen(cmIfEQ);
   lexLE : Gen(cmIfGT);
   lexLT : Gen(cmIfGE);
   lexGE : Gen(cmIfLT);
   lexGT : Gen(cmIfLE);
   end;
end;
```

Генерация кода для операторов

Распознаватели операторов присваивания и вызова процедуры, операторов **IF** и **WHILE** вызываются из уже написанной при рассмотрении контекстного анализа процедуры Statement (см. листинг 3.37). Теперь запрограммируем эти распознаватели. В их задачу входит синтаксический и контекстный анализ соответствующих операторов и генерация кода для них.

Трансляция оператора присваивания

Транслятор оператора присваивания строится в точном соответствии с синтаксисом этой конструкции:

```
Переменная ":=" Выраж.
```

Для анализа и генерации кода для переменной и выражения вызываются распознаватели Variable и IntExpression (листинг 3.57). При трансляции переменной должен быть сформирован код, оставляющий на вершине стека адрес этой переменной. Транслятор выражения формирует команды, вычисляющие выражение и оставляющие его значение на стеке. Напомню, что IntExpression (см. листинг 3.49) не только выполняет синтаксический анализ и генерацию кода для выражения, но и проверяет, имеет ли выражение целый тип.

Процедуре AssStatement остается сформировать команду SAVE, которая запишет значение, находящееся на вершине стека по адресу, находящемуся под вершиной.

Листинг 3.57. Трансляция оператора присваивания

```
(* Переменная ":=" Выраж *)
procedure AssStatement;
begin
   Variable;
   if Lex = lexAss then begin
      NextLex;
      IntExpression;
      Gen(cmSave);
      end
   else
      Expected('":="')
end;
```

Таким образом, код, формируемый для присваивания вида `v := E`, где `v` — переменная, а `E` — выражение, имеет следующую структуру:

```
Av
Код для E
SAVE
```

Здесь `Av` обозначает адрес переменной `v`. Транслятор переменной — процедура `Variable` – показана в листинге 3.58.

Листинг 3.58. Трансляция переменной

```
(* Переменная = Имя. *)
procedure Variable;
var
   X : tObj;
begin
   if Lex <> lexName then
      Expected('имя')
   else begin
      Find(Name, X);
      if X^.Cat <> catVar then
         Expected('имя переменной');
      GenAddr(X);
      NextLex;
   end;
end;
```

В языке «О» в роли переменной может использоваться только имя. Выполняется поиск имени в таблице имен, проверка того, что оно принадлежит переменной и формирование команды, представляющей адрес этой переменной. В качестве параметра процедуре `GenAddr`,

ответственной за генерацию адреса, передается ссылка на переменную в таблице имен (X).

Трансляция вызовов стандартных процедур

Анализатор операторов Statement (см. листинг 3.37), встретив имя процедуры, определяет ее порядковый номер (P) по таблице имен и вызывает процедуру CallStatement (листинг 3.59), передав ей этот номер в качестве параметра. Текущей лексемой в момент вызова является имя процедуры. CallStatement начинает с того, что пропускает эту лексему. Предусмотренная при этом проверка с помощью Check, по сути, является фиктивной, поскольку вызывающая программа уже определила, что имя действительно принадлежит процедуре. Вызов Check вместо NextLex использован здесь лишь для наглядности.

Листинг 3.59. Транслятор операторов вызова процедуры

```
 (* Имя [ "(" Параметр {"," Параметр} ")" ] *)
procedure CallStatement(P : integer);
begin
   Check(lexName, 'имя процедуры');
   if Lex = lexLPar then begin
      NextLex;
      StProc(P);
      Check(lexRPar, '")"');
      end
   else if P in [spOutLn, spInOpen] then
      StProc(P)
   else
      Expected('"("');
end;
```

Вызов процедуры без параметров, а их в языке «О» две: Out.Ln и In.Open, может содержать пустую пару скобок или не содержать скобок вообще. Если же отсутствуют скобки при вызове процедур, которые обязаны содержать параметры, сообщается об ошибке. Замечу, что значение P здесь всегда корректно и соответствует одной из предусмотренных в языке «О» стандартных процедур.

Трансляция списка фактических параметров и генерация кода для конкретных процедур выполняется подпрограммой StProc. Как и при обработке вызовов стандартных функций (см. листинг 3.50), спи-

сок фактических параметров каждой процедуры транслируется индивидуально (листинг 3.60).

Листинг 3.60. Трансляция фактических параметров стандартных процедур

```
procedure StProc(P: integer);
var
   c : integer;
begin
   case P of
   spDEC:
      begin
         Variable;
         Gen(cmDup);
         Gen(cmLoad);
         if Lex = lexComma then begin
            NextLex;
            IntExpression;
            end
         else
            Gen(1);
         Gen(cmSub);
         Gen(cmSave);
      end;
   spINC:
      begin
         Variable;
         Gen(cmDup);
         Gen(cmLoad);
         if Lex = lexComma then begin
            NextLex;
            IntExpression;
            end
         else
            Gen(1);
         Gen(cmAdd);
         Gen(cmSave);
      end;
   spInOpen: { Пусто };
   spInInt:
      begin
         Variable;
         Gen(cmIn);
         Gen(cmSave);
```

```
      end;
   spOutInt:
      begin
         IntExpression;
         Check(lexComma, '","');
         IntExpression;
         Gen(cmOut);
      end;
   spOutLn:
      Gen(cmOutLn);
   spHalt:
      begin
         ConstExpr(c);
         GenConst(c);
         Gen(cmStop);
      end;
   end;
end {case};
```

Трансляция оператора DEC(v), где v — переменная, порождает такой код:

```
Av
DUP
LOAD
1
SUB
SAVE
```

Если вызов DEC имеет вид DEC(v, n), где n — выражение, то генерируются команды:

```
Av
DUP
LOAD
Код для n
SUB
SAVE
```

Аналогично транслируется вызов процедуры INC. Процедура In.Open включена в язык «О» исключительно для совместимости с существующими реализациями Оберона и никакого кода не порождает. Трансляция In.Int, Out.Int и Out.Ln представляется очевидной. Аргументом процедуры HALT может быть константное выражение, значение которого остается на стеке после остановки программы

и выводится в качестве кода возврата по окончании работы виртуальной машины (см. листинг 3.44).

Трансляция оператора WHILE

Синтаксический и контекстный анализ конструкции

```
WHILE ЛогическоеВыражение DO
   ПоследовательностьОператоров
END
```

очень прост. Необходимые проверки выполняются несколькими вызовами процедур:

```
Check(lexWHILE, 'WHILE');
BoolExpression;
Check(lexDO, 'DO');
StatSeq;
Check(lexEND, 'END');
```

Генерация сводится к формированию двух команд перехода: условного, выполняемого за пределы цикла, если условие цикла ложно, и безусловного перехода назад на начало цикла, выполняемого по завершении последовательности операторов. Структура кода должна быть такой:

```
WhilePC: Код для логического выражения
         завершается условным переходом:
         EndPC   ; адрес перехода вперед
         IF..    ; команда условного перехода
CondPC:
         Код для
         последовательности
         операторов

         WhilePC ; адрес начала цикла
         GOTO

EndPC:   …
```

В этой схеме использованы метки, отделенные от обозначений команд двоеточием — обычное решение для языков ассемблера. Метки соответствуют адресам тех команд, на которые ссылаются. Значением метки `WhilePC` является адрес первой команды, вычисляющей логическое выражение; `CondPC` — это адрес участка кода, соответствующего последовательности операторов; `EndPC` — адрес участка про-

граммы, следующего за циклом. Во время компиляции WhilePC, CondPC и EndPC — это значения программного счетчика времени компиляции (переменная PC модуля OGen) перед началом трансляции цикла, после трансляции логического выражения и после трансляции END соответственно.

Для формирования команд выполняется следующее:

1. Перед трансляцией цикла (логического выражения) запоминается в локальной переменной WhilePC текущее значение программного счетчика.
2. Значение PC по окончании трансляции логического выражения запоминается в локальной переменной CondPC.
3. После трансляции последовательности операторов генерируется адрес перехода назад на начало цикла, равный значению WhilePC и команда GOTO. Можно считать, что эти команды порождаются как результат трансляции слова END, завершающего цикл.
4. По адресу, равному CondPC-2 записывается текущее значение PC (в предыдущих рассуждениях было обозначено EndPC). Тем самым завершается формирование перехода вперед.

Перечисленные действия выполняет процедура WhileStatement, приведенная в листинге 3.61. За генерацию перехода вперед с фиктивным адресом отвечает процедура BoolExpression. Запись текущего значения PC по адресу CondPC-2, завершающая генерацию этого перехода, выполняется вызовом Fixup(CondPC) (fixup — адресная привязка — общепринятый термин, обозначающий именно то, что и поручено этой процедуре, — фиксацию значений адресов, которые не могли быть определены раньше).

Листинг 3.61. Трансляция оператора цикла

```
procedure WhileStatement;
var
   WhilePC  : integer;
   CondPC   : integer;
begin
   WhilePC := PC;
   Check(lexWHILE, 'WHILE');
   BoolExpression;
   CondPC := PC;
   Check(lexDO, 'DO');
   StatSeq;
   Check(lexEND, 'END');
```

```
      Gen(WhilePC);
      Gen(cmGOTO);
      Fixup(CondPC);
   end;
```

Поскольку операторы цикла могут быть вложенными, принципиально важно, что `WhilePC` и `CondPC` локальны в `WhileStatement`.

Трансляция оператора **IF**

Подход к трансляции оператора **IF** сходен с методами, использованными при компиляции цикла **WHILE**. Но **IF** может порождать только переходы вперед, к тому же, количество таких переходов из-за присутствия частей **ELSIF** заранее неизвестно.

Правила записи конструкции задаются формулой:

```
IF Выраж THEN
   ПослОператоров
{ELSIF Выраж THEN
   ПослОператоров}
[ELSE
   ПослОператоров]
END
```

а синтаксический и контекстный анализ выполняются следующим образом:

```
Check(lexIF, 'IF');
BoolExpression;  {Логическое выражение}
Check(lexTHEN, 'THEN');
StatSeq;
while Lex = lexELSIF do begin
   NextLex;
   BoolExpression; {Логическое выражение         }
   Check(lexTHEN, 'THEN');
   StatSeq;        {Последовательность операторов}
end;
if Lex = lexELSE then begin
   NextLex;
   StatSeq;        {Последовательность операторов}
end;
Check(lexEND, 'END');
```

Генерация кода должна обеспечить формирование по одному условному переходу на каждое условие (логическое выражение). Каждый такой переход должен быть направлен в обход той последо-

вательности операторов, которая выполняется при истинности данного условия (рис. 3.9). Сами команды перехода формируются при трансляции логических выражений. Задача транслятора **IF** — лишь зафиксировать (с помощью Fixup) адреса этих переходов.

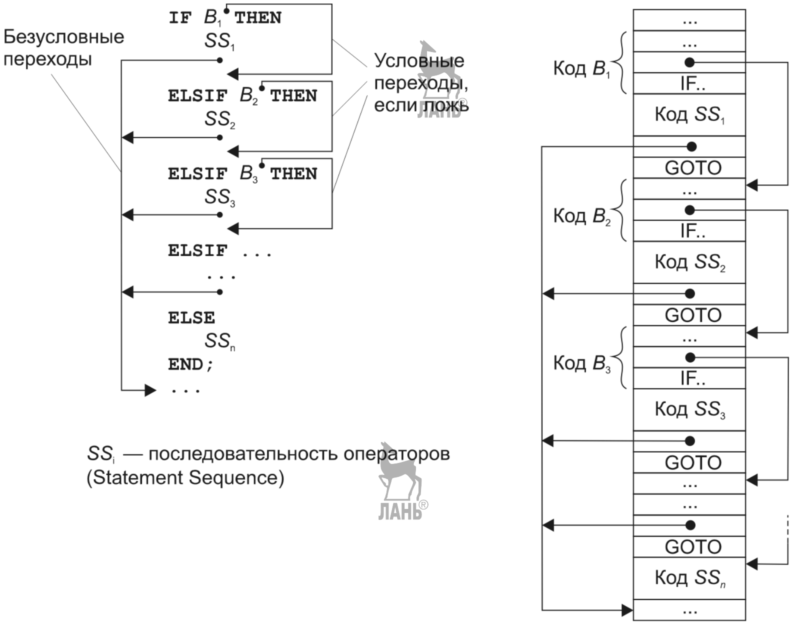


Рис. 3.9. Схема трансляции оператора **IF**

Безусловные переходы должны быть сгенерированы после кода каждой последовательности операторов кроме последней. Адрес всех переходов один и тот же, все они должны выполняться на команду, следующую за оператором **IF**. Оказывается, что адрес **заранее неизвестного количества** безусловных переходов не может быть определен до окончания трансляции всего оператора, а при трансляции завершающего **END** нужно занести текущее значение счетчика команд PC в несколько ячеек. Адреса этих ячеек должны запоминаться по ходу трансляции. Для этих целей можно было бы использовать массив или список, но мы поступим по-другому.

Последовательность адресов ячеек, в которые нужно занести адрес безусловного перехода вперед, будем хранить в самих этих ячейках! Каждая следующая (с большим адресом) ячейка будет хранить адрес предыдущей. В самую первую ячейку (не имеющую предшествующей) запишем 0. Адрес последней (с наибольшим адресом) ячейки будем хранить в локальной переменной (`LastGOTO`). На самом деле в каждой из упомянутых ячеек и в переменной `LastGOTO` будет храниться не сам адрес, а его значение, увеличенное на 2. Это позволяет унифицировать подходы, запоминая места как условных (с помощью `CondPC`), так и безусловных переходов (с помощью `LastGOTO`) **после** формирования самих команд перехода.

Листинг 3.62. Трансляция условного оператора

```
procedure IfStatement;
var
   CondPC   : integer;
   LastGOTO : integer;
begin
   Check(lexIF, 'IF');
   LastGOTO := 0;     {Предыдущего перехода нет        }
   BoolExpression;
   CondPC := PC;      {Запомн. положение усл. перехода}
   Check(lexTHEN, 'THEN');
   StatSeq;
   while Lex = lexELSIF do begin
      Gen(LastGOTO); {Фиктивный адрес, указывающий   }
      Gen(cmGOTO);   {на место предыдущего перехода. }
      LastGOTO := PC;{Запомнить место GOTO           }
      NextLex;
      Fixup(CondPC); {Зафикс.адрес условного перехода}
      BoolExpression;
      CondPC := PC;  {Запомн. положение усл. перехода}
      Check(lexTHEN, 'THEN');
      StatSeq;
   end;
   if Lex = lexELSE then begin
      Gen(LastGOTO); {Фиктивный адрес, указывающий  }
      Gen(cmGOTO);   {на место предыдущего перехода }
      LastGOTO := PC;{Запомн. место последнего GOTO }
      NextLex;
      Fixup(CondPC); {Зафикс. адрес условн. перехода}
      StatSeq;
      end
```

```
      else
         Fixup(CondPC); {Если ELSE отсутствует        }
      Check(lexEND, 'END');
      Fixup(LastGOTO);  {Направить сюда все GOTO      }
   end;
```

Процедура Fixup (листинг 3.63) может обеспечить как фиксацию адреса одиночного перехода, так и заполнение цепочки ячеек, когда это требуется при компиляции **IF**. Фиксация одиночного перехода оказывается частным случаем формирования цепочки адресов при условии, что при генерации кода для одиночного перехода вперед в качестве фиктивного адреса был записан 0. Это требование соблюдается (см. листинги 3.52, 3.56). Можно заметить, что значение 0 не может быть ссылкой на предыдущий переход, поскольку это означало бы, что адрес такого перехода располагается в ячейке с адресом –2, которой не существует.

Листинг 3.63. Адресная привязка

```
procedure Fixup(A: integer);
var
   temp: integer;
begin
   while A > 0 do begin
      temp := M[A-2];
      M[A-2] := PC;
      A := temp;
   end;
end;
```

В качестве входного параметра процедуре Fixup передается адрес A, указывающий на место последней (или просто одной) команды перехода (сам адрес перехода располагается в ячейке A-2). Fixup записывает на место фиктивных адресов текущее значение программного счетчика времени компиляции (переменная PC модуля OGen). Процедуру Fixup имеет смысл разместить в модуле OGen.

Завершение генерации

Завершающим шагом в формировании машинного кода будет генерация команд, останавливающих программу. При распознавании точки в конце транслируемого модуля запишем в последовательность машинных команд константу 0, которая будет служить кодом нормального завершения программы, и команду STOP. Необходимые действия по генерации добавляются в распознаватель модуля (листинг 3.55).

Назначение адресов переменным

Машинный код для программы сформирован. Известен его размер. Он равен значению программного счетчика времени компиляции (PC), которое счетчик примет после генерации последней команды программы — команды STOP.

Теперь самое время вспомнить о необходимости назначения адресов переменным, поскольку только по завершении генерации команд, то есть после формирования команды STOP, стал известен адрес той ячейки памяти, начиная с которой можно разместить переменные. Этот адрес равен значению PC.

Всем переменным программы назначим последовательные адреса, начиная с текущего значения PC. Для каждой переменной языка «О» требуется ровно одна ячейка памяти. Перечень всех переменных можно получить, просматривая таблицу имен.

Назначив переменной адрес, необходимо занести этот адрес во все места машинной программы, где должна быть ссылка на эту переменную. Список адресов ячеек памяти, в которые должен быть записан адрес данной переменной, можно хранить в самих этих ячейках, подобно тому, как при генерации кода для **IF** запоминалось местоположение безусловных переходов (рис. 3.10).

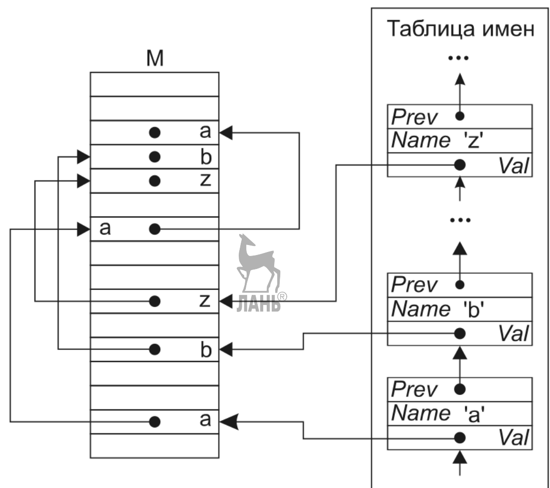


Рис. 3.10. Схема адресной привязки переменных

В роли указателя на цепочку адресов для данной переменной используем поле `Val` записи об этой переменной в таблице имен. В каждую ячейку, в которую надо записать адрес переменной, будем в ходе генерации команд помещать адрес предыдущей такой же ячейки. Эту работу выполнит процедура `GenAddr` (листинг 3.64), вызовы которой уже неоднократно использовались, но реализация до настоящего момента оставалась тайной.

Листинг 3.64. Формирование цепочки ячеек под видом генерации адреса

```
procedure GenAddr(X: tObj);
begin
   Gen(X^.Val);{В текущую ячейку адрес предыдущей + 2}
   X^.Val := PC+1; {Адрес+2 = PC+1}
end;
```

Чтобы унифицировать подходы, будем хранить в образующих цепочку ячейках не сами адреса, а их значения, увеличенные на 2. Это позволит для прохода по цепочкам с целью окончательной фиксации адресов использовать ту же программу адресной привязки `Fixup` (см. листинг 3.63).

В качестве параметра процедура `GenAddr` получает ссылку `X` на запись о переменной в таблице имен. Значение `X^.Val`, содержащее увеличенный на 2 адрес предыдущей ячейки, где должна быть ссылка на ту же переменную, генерируется в качестве очередной команды. Увеличенный на 2 адрес ячейки, куда эта команда попала, становится новым значением поля `Val` записи о переменной.

Чтобы предлагаемая схема работала, необходимо в качестве начального значения поля `Val` в записи таблицы имен для каждой переменной взять 0. Это обеспечивается при добавлении нового имени процедурой `NewName` (см. листинг 3.23).

Для размещения переменных в памяти нужно по завершении генерации кода просмотреть таблицу имен, каждой имеющейся там переменной назначить адрес на единицу больший адреса предыдущей и записать этот адрес во все ячейки цепочки, относящейся к данной переменной. Эти действия выполнит процедура `AllocateVariables` (allocate variables — разместить переменные) модуля `OGen`. Ее вызов можно видеть в листинге 3.65, а реализацию в листинге 3.66.

Листинг 3.65. Завершение генерации кода

```
procedure Module;
var
   ModRef: tObj; {Ссылка на имя модуля в таблице}
begin
   Check(lexMODULE, 'MODULE');
   …
   Check(lexDot, '"."');
   Gen(0);             {Код возврата}
   Gen(cmStop);        {Команда останова}
   AllocateVariables; {Размещение переменных}
end;
```

Первой переменной назначается адрес, равный значению PC после генерации кода. Далее PC продолжает увеличиваться и, как всегда, используется в роли программного счетчика периода компиляции, обозначая теперь адрес, назначаемый очередной переменной.

Листинг 3.66. Размещение переменных в памяти (назначение адресов)

```
procedure AllocateVariables;
var
   VRef: tObj; {Ссылка на переменную в таблице имен}
begin
   FirstVar(VRef);          {Найти первую переменную}
   while VRef <> nil do begin
      if VRef^.Val = 0 then
         Warning(
         'Переменная '+VRef^.Name+' не используется'
         )
      else begin
         Fixup(VRef^.Val); {Адр. привязка переменной}
         PC := PC + 1;
      end;
      NextVar(VRef);     {Найти следующую переменную}
   end;
end;
```

Изменение значения поля Val в записях о переменных происходит только при вызове GenAddr, то есть в момент трансляции фактического использования переменной в программе. Это позволяет обнаруживать неиспользованные переменные, не распределять для них память и выдавать соответствующие предупреждения, если значение Val осталось равным 0.

Просмотр таблицы имен с целью поиска в ней записей о переменных выполняется процедурами `FirstVar` и `NextVar` (листинг 3.67) модуля `OTable`. `FirstVar` служит для поиска первой переменной, `NextVar` — следующих. Результат поиска — ссылка `VRef` на запись о переменной.

Листинг 3.67. Просмотр таблицы имен с целью поиска переменных

```
var
   CurrObj: tObj;

procedure FirstVar(var VRef : tObj);
begin
   CurrObj := Top;
   NextVar(VRef);
end;

procedure NextVar(var VRef : tObj);
begin
  while(CurrObj<>Bottom) and (CurrObj^.Cat<>catVar) do
    CurrObj := CurrObj^.Prev;
  if CurrObj = Bottom then
    VRef := nil
  else begin
    VRef := CurrObj;
    CurrObj := CurrObj^.Prev;
  end
end;
```

Если переменная не найдена, выходной параметр `VRef` будет равен **`nil`**. Процедуры `FirstVar` и `NextVar` используют глобальную (в секции реализации модуля `OTable`) переменную `CurrObj` — указатель на очередной объект таблицы имен.

Трансляция процедур

В языке «О» нет процедур. Есть только возможность вызывать стандартные процедуры и процедуры-функции. Мы не будем «официально» вводить процедуры в язык и не будем переписывать компилятор для языка «О с процедурами». Рассмотрим, не вдаваясь излишне в детали, лишь принципиальные моменты, возникающие при компиляции описаний и вызовов процедур и процедур-функций. Этого должно оказаться достаточно, чтобы при желании самостоятельно реализовать соответствующие механизмы в трансляторе.

Будем считать, что язык «О» дополнен процедурами в соответствии с правилами Оберона при следующих упрощениях:

- Параметры процедур могут быть только целого типа.
- Процедуры не могут быть вложенными.

Будем рассматривать различные варианты трансляции процедур, продвигаясь от простых частных случаев к общему.

Расширенный набор команд виртуальной машины

Для поддержки процедур введем в систему команд ОВМ несколько дополнительных инструкций, обходиться без которых было бы, если не невозможно, то затруднительно. Новые команды, во многом подобные инструкциям реальных процессоров, представлены в таблице 3.4.

Таблица 3.4. Дополнительные команды ОВМ

Код	Обозн.	Название	Стек	Действие
Вызов процедуры и возврат из нее				
-24	CALL	Вызов	A → PC (PA → RA)	M[SP] ↔ PC
-25	RET	Возврат	P0, P1, …, Pn-1, RA, n →	PC := RA; SP := SP+n+2
Выделение и освобождение памяти в стеке				
-26	ENTER	Выделение	n → x1, x2, … xn	SP := SP–n+1
-27	LEAVE	Освобождение	x1, x2, … xn, n →	SP := SP+n+1
Операции с регистром базы				
-28	GETBP	Получить BP	→ BP	M[SP] := BP
-29	SETBP	Установить BP	A →	BP := A
Загрузка и сохранение локальных переменных				
-30	LLOAD	Загрузка лок.	A → M[BP-A]	M[SP]:= M[BP-A]
-31	LSAVE	Сохранение лок.	A, V →	M[BP-A] := V
Получение указателя стека				
-32	SP	Получить SP	→ SP	M[SP-1]:=SP; SP:=SP–1

Кроме дополнительных инструкций в архитектуру ОВМ добавлен регистр базы BP (base pointer — указатель базы). Он предназначается для относительной адресации локальных переменных. Назначение конкретных команд будет разъясняться по мере их использования.

Процедуры без параметров и локальных переменных

Рассмотрим описание процедуры `Pr`, не имеющей формальных параметров и локальных переменных:

```
PROCEDURE Pr;
BEGIN
    …
END Pr;
```

Такая процедура может оперировать только глобальными переменными. Её операторы транслируются как обычно, поэтому сейчас для нас несущественны и заменены многоточием.

Трансляция заголовка процедуры

При обработке заголовка (**PROCEDURE** `Pr;`) компилятор должен поместить идентификатор процедуры (`Pr`) в таблицу имен, в пространство имен модуля. В поле значения имени `Val` следует записать текущее значение счетчика команд `PC` — это будет адрес процедуры, то есть номер ячейки, начиная с которой в памяти будут располагаться относящиеся к процедуре машинные команды. Обозначим этот адрес `PA` — адрес процедуры (procedure address).

Трансляция вызова процедуры

Оператор вызова процедуры в рассматриваемом случае состоит из имени этой процедуры:

```
Pr;
```

Код, порождаемый для этого оператора, должен обеспечить переход по адресу процедуры `PA` с последующим возвратом на оператор, следующий за вызовом процедуры. Именно для такой работы и предназначена команда `CALL` — вызов подпрограммы, выполняющая «переход с возвратом».

Команда `CALL` присваивает программному счетчику значение, взятое с вершины стека, обеспечивая тем самым переход на команду по этому адресу. При этом на вершину стека записывается значение, которое имел перед этим программный счетчик. Напомню, что при исполнении данной машинной команды программный счетчик уже указывает на следующую. В результате выполнения `CALL` на вершине стека окажется адрес команды, следующей за ней. Это и будет адрес возврата. Будем обозначать его `RA` (return address — адрес возврата).

По завершении работы процедуры этот адрес, сохраненный на стеке, позволит выполнить возврат к командам вызывающей программы.

Можно заметить, что действие команды CALL сводится к обмену значений PC и значения с вершины стека, что и отражено в таблице 3.4.

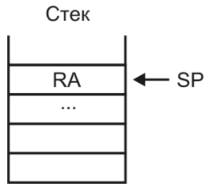
Таким образом, при трансляции вызова процедуры без параметров и локальных переменных компилятор должен поместить в генерируемую последовательность команд адрес процедуры PA и команду CALL:

Исходный текст	Машинный код
Pr;	PA ; Адрес процедуры CALL ; Переход с возвратом

Вход в процедуру и возврат из процедуры

После вызова процедуры, то есть перехода по адресу PA, на вершине стека находится адрес возврата. При таком состоянии стека процедура начинает работать.

В рассматриваемом простейшем случае нет необходимости формировать какие-либо специальные команды, которые будут выполняться при входе в процедуру.



Операторы, составляющие процедуру, транслируются как обычно.

Каждый оператор оставляет стек в том же состоянии, что и получил. Поэтому после выполнения последнего оператора процедуры стек будет в том же состоянии, что и сразу после ее вызова. На вершине стека будет располагаться RA. Чтобы обеспечить возврат по адресу, находящемуся на вершине стека, достаточно сгенерировать в конце кода процедуры (при распознавании **END**) команду GOTO.

Исходный текст	Машинный код
PROCEDURE Pr;	PA) …
BEGIN	…
…	…
END Pr;	GOTO

Процедуры с параметрами-значениями без локальных переменных

Рассмотрим теперь процедуру с *n* параметрами-значениями следующего вида:

```
PROCEDURE Pr(P0, P1, …, Pn-1: INTEGER);
BEGIN
   …
END Pr;
```

Первый вопрос, который нужно решить — это способ размещения параметров процедуры в памяти.

Распределение памяти для формальных параметров и локальных переменных

Можно предложить несколько способов размещения в памяти параметров и локальных переменных процедур.

Статическое распределение памяти

В этом случае каждому параметру каждой процедуры и каждой локальной переменной[79] может быть отведена собственная постоянная ячейка памяти. При этом может использоваться тот же способ назначения адресов, который применялся для глобальных переменных. Ниже приведен эскиз модуля, содержащего две процедуры, и распределение памяти под переменные и параметры.

Исходный текст	Машинный код
`MODULE M;`	
`VAR X : INTEGER;`	
`PROCEDURE Pr1(P1: INTEGER);`	
`BEGIN … END Pr1;`	…
`PROCEDURE Pr2(P2: INTEGER);`	
`BEGIN … END Pr2;`	…
`BEGIN`	
`   …`	…
`END M.`	STOP
	X
	P1
	P2

[79] С точки зрения распределения памяти и организации доступа к ней нет существенной разницы между формальными параметрами и локальными переменными, особенно если речь идет о параметрах-значениях.

При статическом распределении параметры процедур и локальные переменные занимают место в памяти даже тогда, когда соответствующая процедура не исполняется.

Такой способ распределения памяти использовался в ранних версиях Фортрана. К его преимуществам можно отнести эффективность доступа к данным и простоту. Недостатки — увеличенный расход памяти и невозможность рекурсии.

Распределение памяти в стеке

Это наиболее распространенный способ размещения локальных переменных и параметров процедур. Он позволяет реализовать рекурсию, не требует выделения памяти под данные тех процедур, которые не исполняются в данный момент. Этот вариант и будет использован.

Трансляция вызова процедуры с параметрами-значениями

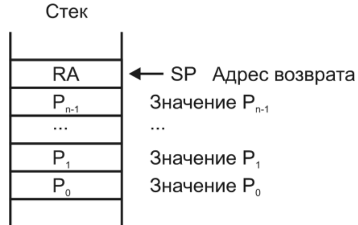
При трансляции вызова процедуры с *n* параметрами-значениями

```
Pr(P0, P1, …, Pn-1);,
```

где P_0, P_1, …, P_{n-1} — выражения, определяющие фактические параметры, компилятор сформирует машинный код для вычисления каждого параметра. Далее генерируется адрес процедуры и команда перехода с возвратом:

```
<Код, вычисляющий P0>
<Код, вычисляющий P1>
…
<Код, вычисляющий Pn-1>
PA   ;Адрес процедуры
CALL ;Вызов
```

Этот код обеспечит при своем выполнении вычисление выражений фактических параметров, их значения останутся в стеке. После этого команда CALL поместит на вершину стека адрес возврата RA:



В таком состоянии стек будет передан вызванной процедуре.

Доступ к параметрам процедуры

Для загрузки и сохранения параметров будут применяться команды локальной загрузки LLOAD и сохранения LSAVE. Они используют относительную адресацию с помощью регистра BP (см. табл. 3.4). Параметр, который эти команды берут с вершины стека, означает величину смещения вверх (в сторону меньших адресов) относительно значения, записанного в базовый регистр BP. Если в регистр базы записать адрес P0, то загрузка параметра Pi может быть выполнена с помощью следующей пары команд:

```
i      ;Относительный адрес
LLOAD ;Команда загрузки по относительному адресу
```

Такое соглашение мы и будем использовать. Регистр базы будет указывать на первую, считая от дна стека, ячейку памяти, относящуюся к процедуре. Участок стека, хранящий данные процедуры, обычно называют кадром (stack frame).

ЛАНЬ®

Трансляция описания процедуры

После распознавания слова **PROCEDURE,** распознавания имени процедуры (например, Pr) и занесения этого имени в пространство глобальных имен транслятор должен открыть с помощью OpenScope новый блок в таблице имен — блок локальных данных процедуры.

При трансляции формальных параметров их идентификаторы заносятся в таблицу имен, а в поле значения Val записывается порядковый номер параметра (от 0 до *n*-1), который будет служить его относительным адресом.

Вход в процедуру

В случае, когда у процедуры имеются параметры и, значит, будет использована адресация относительно регистра базы, компилятор должен сгенерировать код, который будет выполняться в начале работы процедуры (при входе в процедуру) и обеспечит установку регистра BP на дно кадра стека[80]. Код, который выполняет подготовительные действия перед выполнением операторов, составляющих процедуру, часто называют *прологом*. Можно считать, что пролог порождается компилятором при распознавании слова **BEGIN**.

ЛАНЬ®

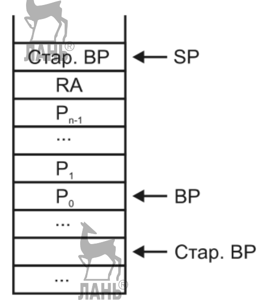
[80] «Установить регистр на…» означает: «занести в регистр значение, равное адресу нужной ячейки».

Поскольку одна процедура может вызываться из другой[81], то при возврате из вызванной процедуры регистр базы должен снова указывать на кадр стека вызывающей. Следовательно, перед тем как `BP` будет установлен в новое значение при входе в процедуру, его старое значение, соответствующее среде вызывающей процедуры, должно быть запомнено с тем, чтобы перед выходом его можно было восстановить. Запоминать старое значение `BP`, естественно, следует на стеке.

Таким образом, в прологе должны быть выполнены (и запрограммированы компилятором) следующие действия:

1. Запомнить на стеке старое (текущее) значение регистра базы `BP`.
2. Записать в регистр базы адрес параметра `P0`.

В результате стек должен оказаться в таком состоянии:



Запоминание старого значения `BP` выполняется просто одной командой `GETBP`. А вот чтобы записать в регистр базы адрес первого параметра процедуры, его придется вычислить, прибавив к значению указателя стека `SP` (он в этот момент указывает на расположенное на вершине стека старое значение `BP`) величину $n+1$ — n параметров плюс ячейка, занятая адресом возврата `RA`. В результате пролог получается таким:

```
GETBP ;Сохранить старое значение BP
SP    ;Значение SP на стек
n+1   ;Это известная компилятору константа
ADD
SETBP ;Установить новое значение BP
```

[81] Речь идет о вложенности *вызовов* процедур (одна процедура может вызывать другую), но не о вложенности самих процедур, которой в нашем языке нет.

Выход из процедуры

Перед возвратом из процедуры стек будет в том же состоянии, как и сразу после выполнения пролога (см. схему выше). Задача эпилога (фрагмента кода, выполняемого непосредственно перед возвратом из процедуры) состоит в следующем:

1. Восстановить старое значение BP (командой SETBP).
2. Перейти по адресу RA (который после выполнения п. 1 находится на вершине стека), удалив из стека *n* параметров. Для выполнения именно таких действий предназначена команда RET (см. табл. 3.4). В качестве параметра RET получает на вершине стека количество снимаемых со стека значений (не считая RA).

Эпилог, который должен быть сгенерирован компилятором:

```
SETBP  ;Восстановить BP
n      ;Известная компилятору константа
RET    ;Возврат из процедуры
```

При выходе из процедуры компилятор также должен закрыть с помощью CloseScope область видимости локальных данных процедуры.

Процедуры с параметрами-значениями и локальными переменными

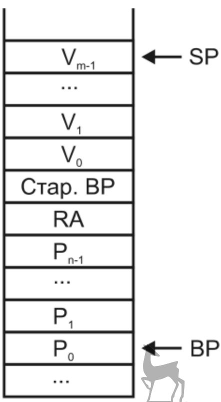
Пусть *n* — число формальных параметров-значений, *m* — количество локальных переменных, а процедура имеет такой вид:

```
PROCEDURE Pr(P0, P1, …, Pn-1: INTEGER);
VAR
   V0, V1, …, Vm-1 : INTEGER;
BEGIN
   …
END Pr;
```

Трансляция заголовка процедуры в этом случае происходит так же, как и в предыдущем.

Трансляция описаний локальных переменных

Задача компилятора при трансляции локальных переменных — разместить их в памяти, т. е. назначить адреса. Значения фактических параметров и адрес возврата записываются в стек при вызове процедуры. Локальные переменные можно разместить в стеке поверх адреса возврата и сохраненного старого значения BP, как показано на схеме.



По мере распознавания идентификаторов локальных переменных компилятор назначает им относительные адреса, начиная со значения n+2. То есть переменная `Vi` (i = 0..n–1) получает адрес i+n+2. Назначенные адреса записываются в поле значения `Val` таблицы имен.

Доступ к локальным переменным для чтения и записи полностью аналогичен доступу к параметрам-значениям и выполняется по относительным адресам с помощью команд `LLOAD` и `LSAVE`.

Вход в процедуру

Резервирование памяти для локальных переменных предусмотрим в прологе. Вначале выполняются те же действия, что в предыдущем случае: сохранение старого и вычисление нового значения `BP`. Затем с помощью команды `ENTER` указатель стека поднимается на m ячеек вверх и тем самым резервируется место для локальных переменных:

```
GETBP ;Сохранить старое значение BP
SP
n+1
ADD
SETBP ;Установить новое значение BP
m     ;Зарезервировать место для
ENTER ;m переменных
```

Выход из процедуры

Последовательность действий, совершаемых при выходе из процедуры, определяется состоянием стека на этот момент. Это состояние

показано на схеме выше. Компилятор должен запрограммировать следующее:

1. Освободить память, занятую локальными переменными. Реализуется командой LEAVE, которая выполняет действия обратные ENTER.
2. Восстановить BP.
3. Возвратиться в вызывающую программу с удалением *n* параметров из стека с помощью команды RET.

Код эпилога получается таким:

```
m     ;Известная компилятору константа
LEAVE ;Удалить переменные
SETBP ;Восстановить BP
n     ;Удаление n параметров и
RET   ;возврат
```

Простейшая оптимизация кода

В ряде частных случаев генерации кода для пролога и эпилога универсальные последовательности команд могут быть заменены более короткими. Некоторые такие оптимизации представлены в таблице 3.5.

Таблица 3.5. Оптимизация кода пролога и эпилога

Неоптимизированный код	Оптимизированный код
1 ENTER	0
1 LEAVE	DROP
0 RET	GOTO

Процедуры-функции с параметрами-значениями и локальными переменными

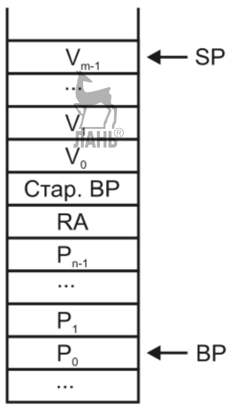
Процедуры-функции вызываются при вычислении выражений. Следуя алгоритму вычисления выражений, представленных в обратной польской записи, функция должна заменять на стеке значения своих аргументов вычисленным значением функции.

Пусть процедура-функция Pr имеет *n* аргументов, использует *m* локальных переменных и ее описание имеет такой вид:

```
PROCEDURE Pr(P0, P1, …, Pn-1: INTEGER): INTEGER;
```

```
VAR
   V₀, V₁, …, Vₘ₋₁ : INTEGER;
BEGIN
   …
END Pr;
```

Сразу после входа и выполнения пролога стек, переданный функции, имеет такой вид:



Возвращая управление вызывающей программе, функция должна поместить вычисленное значение на дно стекового кадра, то есть в ячейку, занятую параметром `P0` (ячейку, адрес которой хранится в `BP`). После возврата из функции эта ячейка должна стать вершиной стека вызывающей программы, то есть не должна освобождаться командой `RET`. Это достигается уменьшением на единицу параметра команды `RET`: вместо пары команд

```
n
RET
```

должна быть сгенерирована последовательность

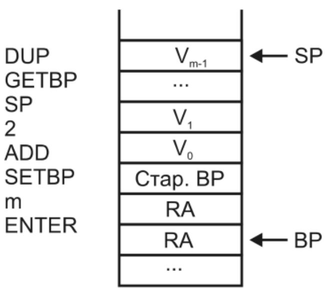
```
n-1
RET
```

Отдельно нужно рассмотреть случай, когда функция не имеет параметров ($n = 0$). Если не принять специальных мер, то в этой ситуации ячейки, куда следует записать вычисленное значение функции, просто не будет: параметр `P0` отсутствует.

Вход в процедуру-функцию

При наличии параметров вход в процедуру-функцию не отличается от обычного.

При $n = 0$ можно было бы в вызывающей программе предусмотреть запись в стек фиктивного фактического параметра, например числа 0. Более изящное решение состоит в том, что в пролог добавляется команда `DUP`, которая дублирует значение адреса возврата `RA`. Верхняя копия `RA` исполняет свою основную роль, нижняя — резервирует место для значения функции. Таким образом, при $n = 0$ (и $m \neq 0$) пролог и стек после его выполнения будут такими:



Выход из процедуры-функции

Ниже в левой колонке приведен код, порождаемый компилятором для эпилога функции, имеющей больше одного аргумента ($n > 1$), в правой колонке — для случая, когда $n = 0$ или $n = 1$.

```
M
LEAVE
SETBP ;Восстановить
BP
n-1   ;На 1 меньше
обычного
RET
```

```
m
LEAVE
SETBP
;Восстановить BP
GOTO  ;возврат
```

В обоих вариантах нижняя ячейка стекового кадра передается вызывающей программе. Можно считать, что приведенный эпилог порождается компилятором при распознавании слова `END`, завершающего процедуру-функцию. В обычной ситуации выполнение функции не должно доходить до `END`, поскольку в этом случае значение функции

не будет определено[82]. Вычисление функции должно завершаться оператором RETURN.

Трансляция оператора RETURN

В собственно процедурах (не функциях) оператор **RETURN** транслируется так же, как **END**, завершающий процедуру. То есть компилятор либо порождает эпилог, либо может генерировать безусловный переход на эпилог, размещенный в конце кода процедуры, либо может не порождать никакого кода, если **RETURN** записан непосредственно перед **END**.

При использовании внутри функции оператор возврата записывается в такой форме:

```
RETURN <Выражение>
```

При его трансляции нужно обеспечить вычисление выражения, запись его значения в ячейку результата, на которую указывает BP, и обычный возврат из функции. Код получается таким:

```
GETBP ;Адрес ячейки результата
<Код для Выражения>
SAVE  ;Сохранить результат
m     ;Если есть
LEAVE ;локальные переменные
SETBP ;Восстановить BP
n-1   ;Если параметров больше одного
RET   ;Возврат
```

Особенность трансляции параметров-переменных

При трансляции вызовов процедур, имеющих параметры-значения и параметры-переменные, компилятор может использовать контекстную информацию для выбора одного из вариантов распознавания фактических параметров. Если очередной формальный параметр — переменная, то соответствующий фактический должен также быть переменной — вызывается распознаватель переменной. Если следует параметр-значение, вызывается распознаватель выражений.

[82] Можно заметить, что при нашей реализации значением функции в этом случае будет либо адрес возврата (если параметров нет), либо значение первого параметра (если он есть). Надеюсь, что никому из программирующих на языке «О с процедурами» не придет в голову использовать эту особенность реализации.

Подстановка фактического параметра вместо параметра-переменной

Примем соглашение. Для формального параметра-переменной в стек при вызове процедуры всегда записывается **абсолютный адрес** фактического параметра.

Пусть заголовок вызываемой процедуры имеет вид:

```
PROCEDURE Pr(VAR V: INTEGER);
```

а вызов этой процедуры таков:

```
Pr(X);
```

Рассмотрим несколько случаев генерации кода для вызова `Pr` в зависимости от того, что такое переменная `X`.

1. `X` — глобальная переменная. Тогда задача решается непосредственно. Для переменной (как в компиляторе языка «О» без процедур) генерируется ее (абсолютный) адрес:

   ```
   X    ;Абсолютный адрес X
   Pr   ;Адрес процедуры Pr
   CALL ;Вызов
   ```

   Здесь и далее адрес (абсолютный или относительный) объекта программы (переменной, процедуры) будем обозначать при записи фрагментов машинного кода просто именем этого объекта. В реальный код адреса глобальных переменных компилятор записывает, используя алгоритм, рассмотренный выше в разделе «Назначение адресов переменным». Адреса процедур, их параметров и локальных переменных извлекаются из таблицы имен, где они хранятся в поле значения `Val` записи о соответствующем объекте.

2. `X` — локальная переменная или формальный параметр-значение. В этом случае абсолютный адрес ячейки, где хранится значение `X`, вычисляется как разность содержимого регистра базы `BP` и относительного адреса `X`.

   ```
   GETBP ;Значение регистра базы
   X     ;Относительный адрес X
   SUB   ;Теперь на стеке абсолютный адрес X
   Pr    ;Адрес процедуры Pr
   CALL  ;Вызов
   ```

3. X — параметр-переменная. Значит, в ячейке с известным относительным адресом, соответствующей формальному параметру X, хранится абсолютный адрес фактического параметра. Используя команду локальной загрузки, получаем на стеке этот абсолютный адрес:

```
X     ;Относительный адрес
LLOAD ;Теперь на стеке абсолютный адрес
Pr    ;Адрес процедуры Pr
CALL  ;Вызов
```

Доступ к параметрам-переменным

Извлечение (загрузка на вершину стека) значения параметра-переменной сводится к получению абсолютного адреса фактического параметра и последующему применению команды абсолютной загрузки LOAD:

```
V     ;Относительный адрес
LLOAD ;Теперь на стеке абсолютный адрес
LOAD  ;На стеке значение
```

Если значение параметру-переменной нужно присвоить, т. е. выполнить действие

```
      V := E,
```

где E — выражение, то вначале должен быть получен абсолютный адрес фактического параметра, затем вычисляется выражение E (его значение остается на стеке), после этого можно применить команду SAVE, использующую абсолютную адресацию:

```
X           ;Относительный адрес
LLOAD       ;Теперь на стеке абсолютный адрес
<Код для E>
SAVE        ;Сохранить
```

Пример программы на языке «О с процедурами»

Для иллюстрации возможностей языка, трансляцию которого мы подробно обсудили, приведу пример законченной программы на «О с процедурами». Программа (листинг 3.68) решает известную головоломку «Ханойские башни»: требуется перенести *n* дисков со стержня 1 на стержень 2, используя стержень 3 в качестве вспомогательного (рис. 3.11).

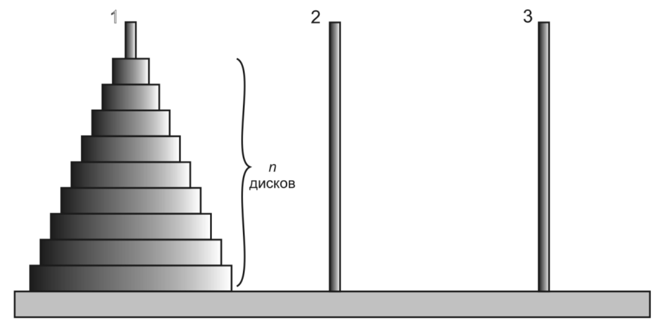


Рис. 3.11. Головоломка «Ханойские башни»

Можно переносить по одному диску. Больший диск никогда не должен находиться поверх меньшего.

Листинг 3.68. «Ханойские башни» на языке «О с процедурами»

```
MODULE Towers;
IMPORT In, Out;

VAR
   n : INTEGER;

PROCEDURE WriteLn(X, Y: INTEGER);
BEGIN
   Out.Int(X, 1); Out.Int(Y, 2); Out.Ln;
END WriteLn;

PROCEDURE Hanoy(n, X, Y, Z: INTEGER);
BEGIN
   IF n > 0 THEN
      Hanoy(n-1, X, Z, Y);
      WriteLn(X, Y);
      Hanoy(n-1, Z, Y, X);
   END;
END Hanoy;

BEGIN
   In.Open;
   In.Int(n);
   Hanoy(n, 1, 2, 3);
END Towers.
```

Программа печатает решение задачи в виде последовательности пар чисел. Первое число пары обозначает стержень, с которого надо взять диск, второе — стержень, на который перенести. Вот пример выполнения программы для $n = 3$:

```
?3
1 2
1 3
2 3
1 2
3 1
3 2
1 2
```

В листинге 3.69 можно видеть код виртуальной машины, который будет сформирован компилятором для программы «Ханойские башни». Участки кода записаны за теми фрагментами исходной программы, которые этот код порождают. Строки исходного кода оформлены как комментарии.

Листинг 3.69. Результат компиляции программы «Ханойские башни»

```
;MODULE Towers;
;IMPORT In, Out;

;VAR
;   n : INTEGER;

;PROCEDURE WriteLn(X, Y: INTEGER);
0)  62
1)  GOTO  ; В обход процедур

;BEGIN
2)  GETBP ; Сохранить стар. BP
3)  SP
4)  3     ; Потому что параметров 2
5)  ADD
6)  SETBP ; Установить BP = SP+3

;   Out.Int(X, 1); Out.Int(Y, 2); Out.Ln;

7)  0     ; Относительный адрес X
8)  LLOAD ; X
9)  1     ; X, 1
10) OUT
11) 1     ; Относительный адрес Y
```

```
12) LLOAD ; Y
13) 2     ; Y, 2
14) OUT
15) OUTLN

;END WriteLn;
16) SETBP ; Восстановить BP
17) 2     ; Удалить 2 параметра
18) RET   ; Возврат из WriteLn

;PROCEDURE Hanoy(n, X, Y, Z: INTEGER);
;BEGIN
19) GETBP ; Сохранить стар. BP
20) SP
21) 5
22) ADD
23) SETBP ; Установить BP = SP+5 (4 параметра)

;   IF n > 0 THEN
24) 0     ; - Относительный адрес n
25) LLOAD ; n
26) 0     ; n, 0
27) 59
28) IFLE

;      Hanoy(n-1, X, Z, Y);
29) 0     ; - Относительный адрес n
30) LLOAD ; n
31) 1     ; n, 1
32) SUB   ; n-1
33) 1     ; - Относительный адрес X
34) LLOAD ; n-1, X
35) 3     ; - Относительный адрес Z
36) LLOAD ; n-1, X, Z
37) 2     ; - Относительный адрес Y
38) LLOAD ; n-1, X, Z, Y
39) 19    ; - Адрес процедуры Hanoy
40) CALL

;      WriteLn(X, Y);
41) 1     ; - Относительный адрес X
42) LLOAD ; X
43) 2     ; - Относительный адрес Y
44) LLOAD ; X, Y
```

```
45) 2      ; - Адрес процедуры WriteLn
46) CALL

;      Hanoy(n-1, Z, Y, X);
47) 0      ; Относительный адрес n
48) LLOAD ; n
49) 1      ; n, 1
50) SUB    ; n-1
51) 3      ; - Относительный адрес Z
52) LLOAD ; n-1, Z
53) 2      ; - Относительный адрес Y
54) LLOAD ; n-1, Z, Y
55) 1      ; Относительный адрес X
56) LLOAD ; n-1, Z, Y, X
57) 19     ; - Адрес процедуры Hanoy
58) CALL

;   END;
;END Hanoy;
59) SETBP ; Восстановить BP
60) 4      ; Удалить 4 параметра
61) RET    ; Возврат

;BEGIN
;   In.Open;
;   In.Int(n);
62) 74     ; Адрес глобальной n
63) IN     ; 74, n
64) SAVE

;   Hanoy(n, 1, 2, 3);
65) 74     ; Адрес n
66) LOAD   ; n
67) 1
68) 2
69) 3
70) 19     ; Адрес процедуры Hanoy
71) CALL

;END Towers.
72) 0
73) STOP
```

Конструкция простого ассемблера

Обсуждая генерацию кода компилятором языка «О», мы постоянно использовали мнемоническую запись команд машинного кода. Однако такая запись рассматривалась лишь как условная, предназначенная для внешнего представления машинных команд. Правила этой записи не были строго определены и время от времени менялись, поскольку не предполагалось, что программы, записанные таким мнемоническим кодом, могут быть непосредственно введены в компьютер и выполнены. Между тем, для этого нет принципиальных препятствий. Нужно лишь формализовать правила записи и написать программу или программы, которые бы выполняли преобразование мнемонического кода в числовой, обеспечивали его загрузку в память ОВМ и запуск. Получится *ассемблер*.

Слово «ассемблер» используют в двух различных смыслах. Во-первых, ассемблер — это программа-транслятор, преобразующая мнемокод в машинные команды. Во-вторых, ассемблером называют сам язык мнемонических команд. Правильнее было бы во втором случае говорить о «языке ассемблера», но сложившуюся практику не отменишь.

Язык ассемблера виртуальной машины

Сформулируем правила языка ассемблера ОВМ. Многие из них будут не новы, поскольку уже использовались при мнемонической записи фрагментов машинного кода.

- Команды виртуальной машины имеют мнемонические обозначения, приведенные в таблицах 3.3 и 3.4. Названия команд всегда записываются заглавными буквами. Заглавные и строчные буквы считаются различными.
- В роли команд могут использоваться целые константы без знака.
- В программу могут быть включены комментарии. Все символы от точки с запятой, включая ее саму, до конца строки являются комментарием и не влияют на смысл программы.
- Строки программы могут быть помечены, в роли меток используются идентификаторы (имена), за которыми следует двоеточие.
- Имена меток могут использоваться в качестве команд для ссылки на ячейки памяти, соответствующие помеченным строкам. Метка-команда означает адрес ячейки и не сопровождается двоеточием.
- Строка программы может содержать не больше одной команды.

- При записи программы могут использоваться пробелы и пустые строки. Пробелы и разделители строк не должны встречаться внутри обозначений команд и меток.

Пример программы на ассемблере

В листинге 3.70 приведена записанная на ассемблере программа нахождения наибольшего общего делителя двух натуральных чисел, эквивалентная программе из листинга 3.40.

Листинг 3.70. Нахождение НОД(X, Y). Программа на ассемблере OBM.

```
;НОД по алгоритму Евклида

      IN  ; X
      IN  ; X, Y

Loop: OVER   ; X, Y, X
      OVER   ; X, Y, X, Y
      Quit
      IFEQ   ; X, Y     На выход (Quit), если X=Y
      OVER   ; X, Y, X
      OVER   ; X, Y, X, Y
      NoSwap
      IFLT   ; X, Y     В обход SWAP, если X>Y
      SWAP   ; Y, X     На вершине большее
NoSwap:
      OVER   ; Min(X, Y), Max(X, Y), Min(X, Y)
      SUB    ; Новое X, Новое Y
      Loop
      GOTO   ; X, Y     На начало цикла

Quit: DROP   ; X        Одно значение было лишним
      0      ; X, 0
      OUT
      OUTLN
      STOP
```

Основное отличие этого текста от приведенного в листинге 3.40 — использование меток и отказ от записи числовых адресов. Это значительно облегчает разработку программы, избавляя от необходимости вести счет номеров ячеек. Возможность применять символические метки — ключевое усовершенствование языка ассемблера. При этом важно заметить, что метки в программе из листинга 3.70 означают в

точности то же, что и соответствующие адреса в программе из листинга 3.40. Можно даже говорить, что значением метки `Loop` является адрес 2, метки `Quit` — 15, а `NoSwap` — 11.

Формальный синтаксис языка ассемблера

Запишем правила языка ассемблера на РБНФ:

```
Программа = Строка {перевод_строки Строка}.
Строка = [Метка][Число|Имя|Код].
```

Это синтаксическая грамматика ассемблера. Простейшие элементы программы: метки, числа, имена и коды команд, как и в случае языка высокого уровня, будем называть лексемами. Их вид определяется лексической грамматикой:

```
Метка = буква {буква | цифра} ":".
Число = цифра {цифра}.
Имя = буква {буква | цифра}.
Код = Имя.
```

Синтаксически понятия «Метка» и «Имя» разделены. Но содержательно они тесно связаны. Метка определяет адрес некоторой точки в программе, имя представляет собой ссылку на этот адрес. В дальнейшем метку с двоеточием будем называть также определяющим вхождением имени, одноименную ссылку без двоеточия — использующим вхождением.

Программирование на ассемблере

Важно понимать, что метки могут употребляться не только для условных и безусловных переходов, но и для обозначения ячеек памяти, когда эти ячейки используются как хранилища данных, то есть в роли переменных.

Рассмотрим простейшую программу, которая складывает два введенных числа (листинг 3.71). Введенные значения вначале записываются в память, а потом извлекаются оттуда и суммируются. Вообще-то, действовать так нет особенной нужды, можно было бы, поместив числа на стек при вводе, тут же их сложить. Но в этом примере нам важно, чтобы программа обращалась к памяти.

Листинг 3.71. Резервирование памяти в программе на ассемблере

```
; Сложение двух чисел
   X    ; Адрес X
   IN   ; Ввести
```

```
    SAVE ; Сохранить по адресу X
    Y    ; Адрес Y
    IN   ; Ввести
    SAVE ; Сохранить по адресу Y
    X
    LOAD ; Загрузить X
    Y
    LOAD ; Загрузить Y
    ADD  ; Сложить
    0    ; Это формат вывода
    OUT  ; Вывести
    OUTLN
    STOP
 X: 0    ; Резервирование места для X
 Y: 0    ; Резервирование места для Y
```

Самое важное здесь — метки X и Y. Они помечают ячейки, в которые записан 0 и которые расположены сразу за ячейкой, где хранится команда STOP. Заносить именно ноль туда, где будут храниться слагаемые, вообще-то совсем не обязательно. Начальные значения никак этой программой не используются и записывать их в ячейки X и Y потребовались лишь для того, чтобы зарезервировать место. Другого способа занять две ячейки сразу за кодом и пометить их наш ассемблер не предоставляется. Вообще-то, в ячейку Y можно было ничего первоначально не записывать. Концовка программы могла быть такой:

```
    …
    STOP
 X: 0    ; Резервирование места для X
 Y:      ; Резервирование места для Y
```

И в этом случае значением метки Y будет адрес той же ячейки.

Программируя на ассемблере, совсем не обязательно размещать переменные сразу за командой STOP. С таким же успехом можно спланировать расположение переменных за любой командой безусловного перехода GOTO или возврата RET.

Наконец, возможно[83] использование нестандартных приемов при размещении переменных. Так, в примере про сложение можно назна-

[83] «Возможно» — не значит «рекомендуется».

чить для хранения X[84] первую ячейку из занимаемых кодом программы, а для Y — вторую (листинг 3.72).

Листинг 3.72. Трюки при резервировании памяти в программе на ассемблере

```
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
;!! Не пробуйте повторить! Опасно !!
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

X: X    ; Адрес X
Y: IN   ; Сначала здесь команда, потом Y
   SAVE ; Сохранить по адресу X
   Y    ; Адрес Y
   IN   ; Ввести
   SAVE ; Сохранить по адресу Y
   X
   LOAD ; Загрузить X
   Y
   LOAD ; Загрузить Y
   ADD  ; Сложить
   0    ; Это формат вывода
   OUT  ; Вывести
   OUTLN
   STOP
```

Дело в том, что команда (адрес X), находящаяся в ячейке, предназначенной для хранения X, будет при работе программы выполнена до того, как в эту ячейку будет записано введенное значение. То же относится и к команде `IN`, на место которой будет записано второе введенное значение.

Получилась программа, которая в ходе исполнения модифицирует себя. Нет никаких причин использовать подобные приемы. Они лишь увеличивают риск ошибок. Экономия нескольких ячеек памяти не стоит того. Отрицательным свойством самомодифицирующейся программы является и ее нереентерабельность[85]. Такая программа может быть выполнена лишь один раз после загрузки в память. Попытка ее повторного выполнения без перезагрузки кода приведет к ошибке.

---

[84] При таком использовании в программе про `X` и `Y` удобней говорить как про переменные, хотя такого понятия в языке ассемблера «официально» не существует.

[85] Реентерабельность — свойство программы, позволяющее повторный вход в нее.

Этот пример приведен здесь лишь для иллюстрации средств языка ассемблера.

Программа, печатающая себя

В качестве еще одного примера программирования на ассемблере рассмотрим программу, печатающую саму себя (листинг 3.73). Речь, правда, не идет об известном сюжете — интроспективной программе, печатающей собственный *исходный* текст. В нашем примере печатается *машинный код*, то есть числовые коды команд, составляющих программу.

Листинг 3.73. Программа, печатающая свой машинный код

```
;Метка Код     Стек
Begin: Begin ; Begin
Loop:  DUP   ; Begin, Begin
       LOAD  ; Begin, M[Begin]
       0
       OUT
       OUTLN ; Begin
       1
       ADD   ; Begin+1
       DUP   ; Begin+1, Begin+1
       End   ; Begin+1, Begin+1, End
       Loop
       IFLE
       DROP
End:   STOP
```

Комментарии, показывающие состояние стека, относятся к первому проходу цикла.

Это еще один пример нетривиального использования меток. Метка `Loop` используется для перехода, `Begin` и `End` отмечают начало и конец того участка кода, который будет напечатан[86]. Адреса, соответствующие `Begin` и `End`, участвуют в арифметических операциях и сравнениях. В таких случаях говорят об использовании адресной арифметики. В языках, предназначенных для надежного программирования, она не разрешена. Ассемблер относится к другому классу.

[86] Можно заметить, что такая программа может печатать не обязательно свой код, а содержимое любого участка памяти от ячейки, помеченной `Begin`, до ячейки, имеющей метку `End`.

Реализация ассемблера

Приступим к разработке программы-ассемблера, т. е. транслятора с языка ассемблера в код ОВМ. Как и в случае с компилятором «О», ассемблер будет помещать формируемый код прямо в память виртуальной машины. После этого полученная программа сразу запускается на исполнение (в случае успешной компиляции, разумеется).

Мы будем использовать классическую двухпроходную схему ассемблирования. Хотя ресурсы современного компьютера позволяют обойтись и одним проходом при трансляции с ассемблера, использование двух проходов для языка, в котором определение имени не обязано предшествовать использованию, оказывается естественней и проще.

Главная программа и вспомогательные модули

Модульная структура ассемблера будет во многом повторять конструкцию компилятора «О». Предусматриваются драйвер исходного текста (`AsmText`), модуль для работы с таблицей имен (`AsmTable`), лексический анализатор (`AsmScan`). Это вспомогательные модули. За собственно ассемблирование будет отвечать `AsmUnit`, объединяющий функции распознавателя и генератора кода. Модуль виртуальной машины используется, разумеется, без всяких изменений. Отвечающий за реакцию на ошибки, модуль `OError` также применен неизменным. Главная программа (листинг 3.74) организует всю работу.

Листинг 3.74. Главная программа ассемблера

```
program OASM;
{Ассемблер виртуальной машины ОВМ}
uses
   AsmText, AsmScan, AsmTable, AsmUnit, OVM;
procedure Init;
begin
   AsmTable.InitNameTable;
   AsmText.OpenText;
   AsmScan.InitScan;
end;

procedure Done;
begin
   AsmText.CloseText;
end;

begin
   WriteLn('Ассемблер виртуальной О-машины');
```

```
   Init;
   AsmUnit.Assemble; {Компилировать}
   OVM.Run;
   Done;
end.
```

Чтобы структура обсуждаемой программы была понятней, я счел необходимым записать вызовы процедур вспомогательных модулей с указанием имени модуля.

Хотя функции драйвера исходного текста, сканера и модуля работы с таблицей остаются в основном теми же, что и в компиляторе языка «О», специфика ассемблера требует внесения изменений в интерфейс этих модулей.

Поскольку транслятор будет двухпроходным, нужна возможность читать исходный текст программы дважды. Для этого в драйвере исходного текста (листинг 3.75) наряду с процедурой `ResetText` предусматривается процедура `OpenText`. `OpenText` должна быть вызвана один раз, чтобы открыть текст (обычно это файл). `ResetText` вызывается, всякий раз, когда необходимо читать исходный текст с начала, то есть перед каждым проходом ассемблера.

Листинг 3.75. Интерфейс драйвера исходного текста

```
unit AsmText;
{Драйвер исходного текста}

interface

const
   chSpace  = ' ';
   chTab    = chr(9);
   chEOL    = chr(13);
   chEOT    = chr(0);

var
   Ch    : char;
   Pos   : integer;
   Line  : integer;

procedure OpenText;
procedure ResetText;
procedure CloseText;
procedure NextCh;

{=====================================================}
```

Сканер для ассемблера (листинг 3.76) оказывается проще, чем для языка высокого уровня, поскольку разновидностей лексем в ассемблере намного меньше.

Листинг 3.76. Интерфейс модуля сканера

```
unit ASMScan;
{Сканер для ассемблера}

interface

const
   NameLen = 31;

type
   tName = string[NameLen];
   tLex = (
      lexLabel,  {Метка}
      lexOpCode, {Код операции}
      lexNum,    {Число}
      lexName,   {Имя}
      lexEOL,    {Конец строки}
      lexEOT     {Конец текста}
   );

var
   Lex      : tLex;     {Текущая лексема}
   Num      : integer;  {Значение числа}
   OpCode   : integer;  {Значение кода операции}
   Name     : tName;    {Строка имени}

   LexPos   : integer;

procedure InitScan;
procedure NextLex;

{====================================================}
```

Под лексемой `lexLabel` подразумевается определяющее вхождение метки, сопровождаемое двоеточием, в то время как `lexName` обозначает использующее вхождение — имя метки, употребленное на правах команды.

Таблица имен в нашем простом ассемблере не имеет блочной структуры. Именами снабжаются лишь метки. Для каждой метки в таблицу имен записывается соответствующий ей адрес.

Листинг 3.77. Интерфейс модуля таблицы имен

```
unit AsmTable;
{Таблица имен}

interface

uses
   AsmScan;

type
   tObj = ^tObjRec;
   tObjRec = record
      Name : tName;
      Addr : integer;
      Prev : tObj;
   end;

{Инициализация}
   procedure InitNameTable;
{Добавить имя в таблицу}
   procedure NewName(Addr: integer);
{Поиск имени}
   procedure Find(var Addr: integer);
{=====================================================}
```

Процедура `NewName` (листинг 3.77) добавляет в таблицу имя, содержащееся в глобальной переменной сканера `Name`, и заносит в запись об этом имени адрес `Addr`, переданный в качестве параметра. Процедура `Find` ищет имя `Name` в таблице и при удачном поиске возвращает в выходном параметре `Addr` соответствующий этому имени адрес.

Ассемблирование

Ассемблирование — это трансляция с языка ассемблера, за которую в нашей программе будет отвечать модуль `AsmUnit`.

Листинг 3.78. Интерфейс основного модуля ассемблера

```
unit AsmUnit; {Модуль ассемблера}

interface

procedure Assemble;

{=====================================================}
```

Первый и второй проходы ассемблера

Замена мнемонических обозначений операций их машинными кодами тривиальна. Обработка числовых констант также не составляет труда. Основная же работа ассемблера связана с трансляцией имен.

Каждое использующее вхождение имени должно быть заменено адресом. Адрес задается местоположением одноименной метки (определяющим вхождением). Поскольку определяющее вхождение может располагаться после использующего, адрес может быть неизвестен к моменту обработки использующего вхождения при первом проходе. Проблему решает использование двух проходов по исходной программе.

На первом проходе обрабатываются только определяющие вхождения и заполняется таблица имен. Транслятор ведет счет машинных команд с помощью программного счетчика периода компиляции, который, как и прежде, обозначим `PC`. Перед первым проходом `PC=0`. Затем счетчик увеличивается на единицу каждый раз, когда в программе встречаются код операции, константа или имя — каждый из этих элементов занимает в машинной программе одну ячейку. Адресом, назначенным данному имени, будет значение `PC` в момент распознавания определяющего вхождения этого имени. Само определяющее вхождение имени (метка) не меняет значения `PC`.

На втором проходе генерируется машинный код, в который записываются адреса, определенные при первом проходе. Адреса извлекаются из таблицы имен. Определяющие вхождения имен на втором проходе игнорируются.

Программирование распознавателя

Как и в случае трансляции с языка высокого уровня, в ассемблере ведущую роль будет играть синтаксический анализатор. Специфика же состоит в том, что предусматриваются два прохода, в каждом из которых нужно решать задачу распознавания. В обоих проходах распознаватель программы действует в соответствии с синтаксисом:

```
Программа = Строка {перевод_строки Строка}.
```

Отличие между проходами состоит в том, что при разборе отдельной строки выполняется различная семантическая обработка. В то же время распознавание следования строк в соответствии с приведенной выше формулой в первом и втором проходах должны выполняться одинаково.

Чтобы не переписывать распознаватель программы дважды, используем немного необычное, но очень подходящее решение: распознающая процедура нетерминала «Программа» будет получать в качестве параметра процедуру-распознаватель строки. Чтобы оформить параметр процедурного типа, определим в секции реализации модуля `AsmUnit` соответствующий тип (листинг 3.79). В этом же месте программного кода поместим описание программного счетчика периода компиляции `PC`.

Листинг 3.79. Глобальные описания в реализации модуля AsmUnit

```
type
   tLineProc = procedure; {Тип распознавателя строки}
var
   PC : integer;
```

Теперь запишем процедуру `Assemble` (листинг 3.80), которая запустит сначала первый, а потом второй проход и напечатает сообщения по завершении компиляции.

Листинг 3.80. Запуск первого и второго проходов ассемблера

```
procedure Assemble;
begin
   Pass(LineFirst);  {Первый проход}
   Pass(LineSecond); {Второй проход}
   WriteLn;
   WriteLn('Компиляция завершена');
   WriteLn('Размер кода ', PC);
   WriteLn;
end;
```

Напомню, что при обнаружении ошибок (например, при первом проходе) вызывается процедура `Error`, которая прерывает выполнение программы.

Оба прохода выполняются вызовом процедуры `Pass` (pass — проход; прогон; просмотр), которой для выполнения первого прохода передается процедура-обработчик строки `LineFirst`, для второго — `LineSecond`.

Структура `Pass` (листинг 3.81) определяется синтаксисом программы. Кроме того, перед каждым проходом вызовами `ResetText` и `NextLex` подготавливается чтение исходного текста программы с начала. Отсчет `PC` при каждом проходе начинается с нуля.

Листинг 3.81. Проход ассемблера

```
(* Программа = Строка { перевод_строки Строка } *)
procedure Pass(Line : tLineProc);
begin
   ResetText;
   NextLex;
   PC := 0;
   Line;    {Распознавание строки}
   while Lex = lexEOL do begin
      NextLex;
      Line; {Распознавание строки}
   end;
   if Lex <> lexEOT then
      Error('Так нельзя');
end;
```

Обработчик строки на первом проходе (процедура `LineFirst`, листинг 3.82) должен при распознавании метки (определяющего вхождения имени) заносить имя в таблицу, назначая в качестве адреса текущее значение `PC`. При распознавании числа, кода операции или использующего вхождения имени достаточно увеличивать `PC`.

Листинг 3.82. Распознаватель-обработчик строки на первом проходе

```
(* Строка = [метка][число|имя|код] *)
procedure LineFirst;
begin
   if Lex = lexLabel then begin
      NewName(PC);
      NextLex;
   end;
   if Lex in [lexName, lexNum, LexOpCode] then begin
      PC := PC + 1;
      NextLex;
   end;
end;
```

Распознаватель строки, предназначенный для второго прохода ассемблера (процедура `LineSecond`, листинг 3.83), работает в соответствии с тем же синтаксисом, но выполняет другие семантические действия. Определяющие вхождения имен (метки с двоеточием) пропускаются, а для кодов операций, чисел и имен генерируется машинный код — по одной команде на каждую из названных лексем.

Листинг 3.83. Распознаватель-обработчик строки при втором проходе

```
 (* Строка = [метка][число|имя|код] *)
procedure LineSecond;
var
   Addr : integer;
begin
   if Lex = lexLabel then
      NextLex;
   case Lex of
   lexName:
      begin
         Find(Addr);
         Gen(Addr);
         NextLex;
      end;
   lexNum:
      begin
         Gen(Num);
         NextLex;
      end;
   lexOpCode:
      begin
         Gen(OpCode);
         NextLex;
      end;
   end;
end;
```

Имя порождает генерацию адреса, который берется из заполненной на первом проходе таблицы имен. При распознавании константы в качестве команды в машинный код записывается она сама. Мнемоническое обозначение операции порождает генерацию соответствующей машинной команды, код которой предоставляет сканер в глобальной переменной `OpCode`.

Для записи команд в машинный код, который помещается в память виртуальной машины, используется процедура `Gen` (листинг 3.84).

Листинг 3.84. Генерация кода

```
procedure Gen(Cmd: integer);
begin
   if PC < MemSize then begin
      M[PC] := Cmd;
```

```
        PC := PC+1;
        end
    else
        Error('Недостаточно памяти');
end;
```

Являясь частью модуля `AsmUnit`, она отвечает и за продвижение программного счетчика периода компиляции `PC` на втором проходе ассемблера.

ЛАНЬ®

Автоматизация построения и мобильность трансляторов

Не обязательно все работы по программированию компилятора, построению, анализу и преобразованию грамматик должны выполняться вручную.

Автоматический анализ и преобразование грамматик

Для ряда задач теории формальных языков и грамматик, существуют алгоритмы, решающие эти задачи. Будучи реализованы в виде специальных программ, они могут быть использованы и используются для анализа и преобразования грамматик при разработке языков программирования и специализированных языков информационных систем. Иногда алгоритма не существует, но автоматизированная попытка поиска решения, тем не менее, может быть предпринята. В качестве примера рассмотрим два сюжета такого рода.

Проверка наличия *LL*(1)-свойства у контекстно-свободной грамматики. Для описания синтаксиса языков программирования в подавляющем большинстве случаев используются КС-грамматики. Если такая грамматика является еще и *LL*(1)-грамматикой, то это дает возможность построить эффективный детерминированный распознаватель языка, работающий, например, методом рекурсивного спуска. Существует алгоритм, позволяющий для произвольной КС-грамматики определить, является ли она *LL*(1)-грамматикой.

Преобразование произвольной КС-грамматики в эквивалентную *LL*(1)-грамматику. Если КС-грамматика не обладает *LL*(1)-свойством, можно предпринять попытку ее преобразования в эквивалентную *LL*(1)-грамматику. Задача такого преобразования является, однако, алгоритмически неразрешимой. Это означает, что не существует и не может существовать алгоритма, решающего ее. Тем не

менее, попытки преобразования могут быть предприняты. Есть программы, которые пытаются выполнять такое преобразование, и основываются на использовании эвристических приемов. Результатом работы такой программы может быть успешное преобразование грамматики либо неудача. В первом случае результатами преобразования можно воспользоваться. Неудача же не обязательно означает, что преобразование невозможно, быть может, его просто не удалось найти.

Достоинство автоматических анализаторов и преобразователей грамматик состоит в том, что в случае достижения с их помощью результата его надежность оказывается весьма высокой, если не абсолютной. Например, при успешном автоматическом преобразовании КС-грамматики к виду $LL(1)$ гарантируется отсутствие искажения языка, то есть эквивалентность грамматик. Проблема может возникнуть, только если сама программа-преобразователь содержит ошибку. При преобразовании вручную шансы ошибиться значительно больше.

Автоматическое построение компилятора и его частей

Другое направление автоматизации при создании транслятора — автоматическая генерация его модулей. Для этого могут быть разработаны специальные программы.

Конструкторы сканеров

Лексический анализатор — один из самых простых блоков транслятора. Правила записи лексем обычно могут быть заданы с помощью автоматной грамматики. Задача разбора для автоматной грамматики эффективно решается конечным автоматом. Нетрудно представить себе программу, которая по заданной автоматной грамматике порождает либо программу-распознаватель (рис. 3.12*а*), либо таблицу переходов конечного автомата (рис. 3.12*б*).

Описания лексем языка, которые поступают на вход конструктора сканеров, обычно записываются с помощью регулярных выражений.

Самым известным конструктором лексических анализаторов является программа Lex. Существуют ее реализации, использующие различные языки программирования, в том числе Си и Паскаль. Как результат своей работы Lex порождает программу на одном из этих языков. При необходимости в порождаемую конструктором программу могут быть встроены части, заранее написанные вручную. Правила записи лексем определяются с помощью несколько расши-

ренной нотации регулярных выражений. Lex позволяет исключать из программы комментарии, однако не предоставляет готовой возможности обработки вложенных комментариев, какие имеются, например, в Обероне.

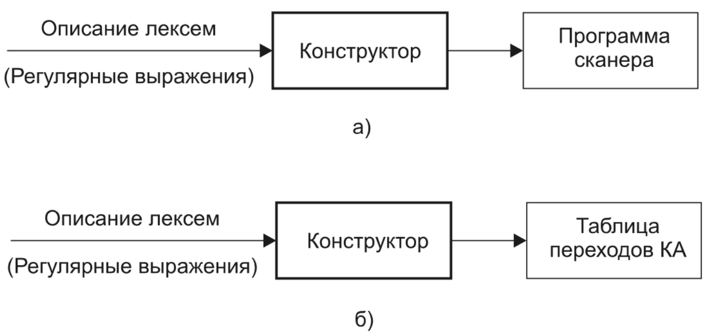


Рис. 3.12. Конструкторы сканеров

Генераторы синтаксических анализаторов

При соблюдении некоторых ограничений на КС-грамматику (например, наличие $LL(k)$ свойства) оказывается возможным автоматически построить эффективный распознаватель для такой грамматики. Программа генератор синтаксических анализаторов, получая на входе описание грамматики, может порождать распознаватель, работающий по таблице, и саму таблицу (рис. 3.13*а*), либо генерировать программу распознавателя, действующего, например, по методу рекурсивного спуска (рис. 3.13*б*).

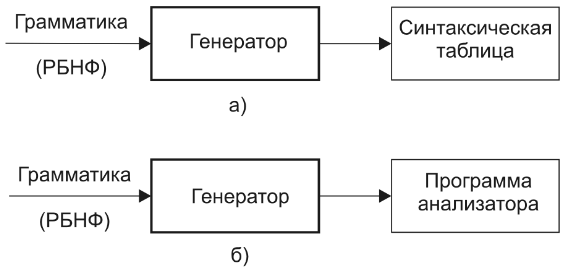


Рис. 3.13. Генераторы синтаксических анализаторов

Табличные распознаватели, применяемые в генераторах анализаторов, могут работать как на основе нисходящих, так и восходящих алгоритмов разбора. Для описания грамматики используются различные варианты БНФ, как правило, расширенные.

Компиляторы компиляторов

Если возможность автоматического порождения лексических анализаторов и синтаксических анализаторов для ограниченного, но вполне содержательного класса КС-грамматик не вызывает сомнений и является обыденной практикой, то полностью автоматическое построение всего компилятора, включая контекстный анализатор и генератор кода — скорее идеал, чем реальная возможность. Между тем, системы, в той или иной мере приближающиеся к этому идеалу, существуют и используются. Их называют компиляторами компиляторов (compiler-compiler). В литературе на русском языке используется термин «система построения трансляторов», сокращенно — СПТ. Общая схема автоматизированной генерации компилятора представлена на рисунке 3.14.

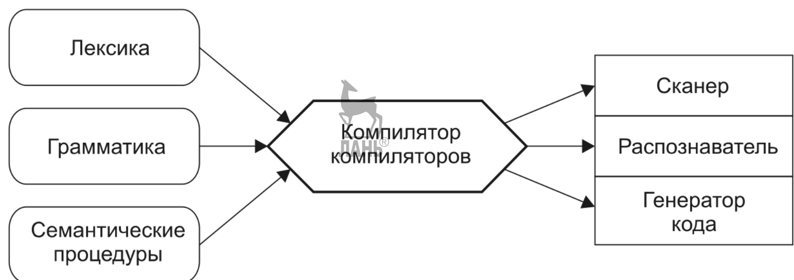


Рис. 3.14. Компилятор компиляторов

В качестве формализма, на основе которого строится описание семантики, обычно используются атрибутные грамматики, введенные в обиход Д. Кнутом. Не надо, однако, представлять дело так, что семантика языка описывается некими формулами, подобными РБНФ. Дело сводится к тому, что разработчик должен написать на используемом в СПТ языке программирования (Си, Паскаль) семантические процедуры, которые обеспечат выполнение контекстного анализа и генерации кода, а СПТ предоставит возможность интегрировать эти процедуры в синтаксический анализатор.

Самым известным компилятором компиляторов является система YACC. Ее название — аббревиатура, происходящая от ироничного Yet Another Compiler Compiler — еще один компилятор компиляторов. К началу 1970-х годов, когда появился YACC, тема автоматического порождения компиляторов была не только популярной, но даже несколько избитой.

Первоначально ориентированный на операционную систему Unix, YACC использовал Си в качестве базового языка. В дальнейшем были разработаны различные варианты YACC-совместимых компиляторов компиляторов, предназначенных для разных операционных систем, использующие отличные от Си базовые языки. Получила известность СПТ Bison[87] — некоммерческая, свободно распространяемая версия YACC.

Компиляторы компиляторов, совместимые с YACC, ориентированные на различные операционные системы и языки программирования, можно найти в Интернете.

YACC интегрируется с конструктором сканеров Lex. На вход СПТ подается описание языка, содержащее фрагменты кода на базовом языке (например, Си), описания лексем и правила трансляции, представляющие собой спецификацию синтаксиса на расширенной БНФ, дополненной возможностью связать терминалы и нетерминалы правил с семантическими процедурами. Результатом работы является текст программы-транслятора (ее основного модуля) на базовом языке.

YACC основан на использовании восходящего *LALR*(1)-распознавателя, работающего по таблице, и ориентированного на *LR*(1)-грамматики — подмножество КС-грамматик, позволяющее определить более широкий класс языков, чем *LL*(1)-грамматики.

Компиляторы компиляторов и программирование вручную

Практика создания компиляторов показывает, что автоматизированная разработка не имеет радикального преимущества в сравнении с программированием вручную. Объясняется это в первую очередь тем, что семантическая составляющая работы компилятора почти не автоматизируется. Что касается лексического и синтаксического анализаторов, то для разумно спроектированного языка трудоемкость

[87] Очевидно, такое название происходит от созвучия YACC и yak — як, ведь бизоны — родственники яков.

реализации этих компонент невелика и составляет очень малую часть всех затрат на разработку.

Программирование компилятора вручную предоставляет бо́льшую гибкость. Получаемая программа имеет ясную структуру, читается намного легче, чем сгенерированная СПТ.

В свою очередь СПТ предоставляют возможность использовать более мощный класс *LR*(1)-грамматик и *LALR*(1)-распознавателей. Такие распознаватели предполагают использование таблиц, построение которых вручную сложно и чревато ошибками.

Использование СПТ может провоцировать усложнение грамматики языка. При ручном программировании неоднозначности грамматики и проблемы детерминированного распознавания часто могут быть решены с помощью несложных контекстных проверок, в то время как анализатор, порождаемый автоматически, требует разрешения таких вопросов уже на уровне синтаксических определений.

Можно предположить также, что написанный вручную код окажется эффективней порожденного компилятором компиляторов.

Некоторые сравнения можно сделать из рассмотрения двух вариантов компилятора языка «О». Один — работающий по методу рекурсивного спуска, другой получен[88] с помощью свободно распространяемой СПТ Turbo Pascal Lex and Yacc Version 4.1. Функционально компиляторы одинаковы. Вариант, полученный с помощью YACC и Lex, использует ряд модулей первого компилятора, в том числе, разумеется, виртуальную машину.

Таблица 3.6. Сравнение компиляторов языка «О»

Характеристика	Рекурсивный спуск	Lex + YACC
Размер исходного кода, написанного вручную		
строк	1368	1064
байт	23 957	25 164
лексем	5149	4837
В том числе сканер		
строк	262	126
байт	5131	2809
лексем	955	534

[88] Работа выполнена В. Ёжкиным.

Продолжение табл. 3.6

Характеристика	Рекурсивный спуск	Lex + YACC
Размер кода полученного компилятора на Паскале		
строк	1368	4650
байт	23 957	85 861
лексем	5149	20 225
Размер исполняемого файла компилятора (байт)	16 656	34 336
Время компиляции тестовой программы[89] (с)	0,37	0,77

Как видно, объем кода, который пришлось написать вручную, отличается в двух вариантах незначительно. Если в качестве единицы измерения использовать число лексем, которое меньше зависит от индивидуального стиля программирования, то при использовании СПТ уменьшение объема ручной работы составило всего 6%, причем вся экономия достигнута на генерации лексического анализатора. Lex действительно позволяет очень быстро получить простой сканер. Так, для выполнения измерений, результаты которых представлены в таблице, с помощью Lex в кратчайший срок был изготовлен сканер, позволяющий подсчитывать число лексем в программах на Паскале и входных файлах Lex и YACC.

Эффективность компилятора, полученного с помощью СПТ, оказалась заметно ниже, чем запрограммированного вручную — он имеет больший размер исполняемого файла и работает медленнее.

Известны примеры крупных разработок, выполненных в 1990-е годы и основанных как на использовании СПТ, так и на программировании вручную. Так, при создании компилятора Си++ в лаборатории открытых информационных технологий ВМиК МГУ (руководитель В. А. Сухомлин, ведущие разработчики Е. А. Зуев, А. Н. Кротов) был использован компилятор компиляторов Bison. Разработка велась на Си и Си++. В то же время компилятор нового языка Си#, созданный компанией Microsoft, также написан на Си++, но с использованием рекурсивного спуска.

[89] 1606 строк на языке «О», компьютер на базе процессора Pentium с тактовой частотой 100 МГц.

Использование языков высокого уровня

Первые трансляторы, появившиеся в 1950-е годы, программировались на машинном языке или на языке ассемблера. В дальнейшем роль языков низкого уровня, как инструментов создания компиляторов, постепенно снижалась. Уже в 1960-е годы некоторые трансляторы программировались на таких языках как Фортран и Алгол-60.

В настоящее время не осталось причин, препятствующих использованию языков высокого уровня при программировании компиляторов и интерпретаторов. Во-первых, существует большой выбор трансляторов для разнообразных языков и программно-аппаратных платформ. Эти языки и эти трансляторы могут служить инструментами для создания новых языков и новых трансляторов.

Во-вторых, мощность компьютеров, на которых исполняются компиляторы[90], настолько возросла, что превышает обычную потребность транслятора в ресурсах, и, следовательно, нет необходимости добиваться максимально возможной эффективности компилятора за счет перехода при его разработке на язык низкого уровня. К тому же, задача трансляции (если не иметь в виду продвинутые методы оптимизации кода) имеет не слишком высокую временную сложность, и ускорение трансляции вряд ли должно быть приоритетным требованием.

В-третьих, достижения в методах оптимизации кода таковы, что хороший оптимизирующий компилятор может создавать программу, почти не уступающую по эффективности написанной вручную. Если есть возможность использовать такой компилятор при разработке, то о программировании на ассемблере следует забыть.

Наконец, трудно себе представить, что компиляторы таких сложных языков, как, например, Си++, Ява, Си# могут быть написаны вручную на ассемблере с разумными затратами и приемлемой надежностью.

Поскольку язык ассемблера для каждой платформы сугубо специфичен, компилятор, написанный на ассемблере, будет обладать низкой мобильностью. Для переноса на другую платформу его придется перепрограммировать полностью или почти полностью. Транслятор же, существующий в виде программы на языке высокого уровня, может быть перекомпилирован для использования на разных системах.

[90] Отнюдь не все современные компьютеры обладают высокой мощностью. Встроенный компьютер, управляющий работой, например, микроволновой печи, не имеет ни быстрого процессора, ни памяти большого объема. Но ведь никто и не пробует запускать компилятор на компьютере печки!

Т-диаграммы трансляторов

С помощью Т-диаграмм рассмотрим возможности трансформации трансляторов, написанных на языках высокого уровня. Транслятор связан с тремя языками: входным (исходным), выходным (целевым, объектным) и инструментальным — языком, на котором написан или существует он сам. Будем как и раньше изображать транслятор в виде Т-диаграммы. Слева на такой диаграмме записывается исходный язык, справа — целевой, внизу — инструментальный (рис. 3.15).

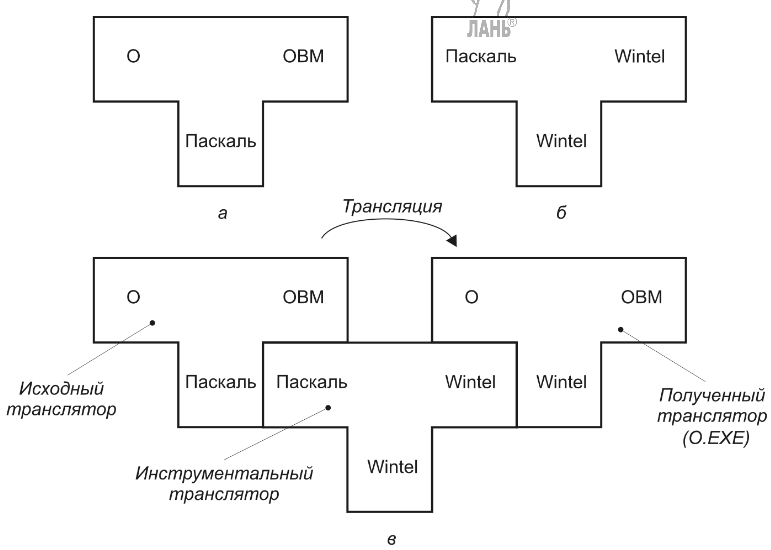


Рис. 3.15. Т-диаграммы:
а — компилятор «О»; *б* — компилятор Паскаля; *в* — трансляция компилятора «О»

На рисунке 3.15*а* можно видеть Т-диаграмму компилятора, который транслирует с языка «О» в код ОВМ и написан на Паскале. Его мы разрабатывали в этой главе. Рисунок 3.15*б* соответствует компилятору языка Паскаль, работающему на платформе «Wintel» (так называют совокупность операционной системы Windows и компьютера на базе процессора Intel). Инструментальный и целевой языки в этом случае — машинный код процессора Intel, к которому добавлены соглашения программного интерфейса Windows. Если компилятор

доступен в виде исполнимого кода, то этот код и можно считать инструментальным языком. Диаграмме с рисунка 3.15*б* соответствуют Turbo Pascal, Free Pascal, Delphi и другие трансляторы Паскаля для Windows. Их и можно использовать для трансляции компилятора языка «О». Эту трансляцию проиллюстрируем с помощью Т-диаграмм, как показано на рисунке 3.15*в*. Компилятор *Паскаль-Wintel-Wintel*[91] транслирует компилятор *О-Паскаль-ОВМ*, преобразуя программу на Паскале в код Wintel, в результате чего получается компилятор *О-Wintel-ОВМ* — программа `O.EXE`.

При создании программ для встроенных микрокомпьютеров (микроконтроллеров), управляющих работой различных технических устройств и систем, используют кросскомпиляторы. Кросскомпилятор, работая на одном компьютере (например, персональном), генерирует машинный код для другого (например, встроенного). Действительно, как уже говорилось, запустить программу-компилятор на микроконтроллере микроволновой печи затруднительно. Другой пример: микроконтроллеры мобильных телефонов обладают памятью и быстродействием, сопоставимыми с возможностями персональных компьютеров недавнего прошлого, но и в этом случае вести разработку программ для телефона на самом телефоне никому не приходит в голову. Пример диаграммы кросскомпилятора можно видеть на рисунке 3.16*а*. Язык Си часто используется для программирования микроконтроллеров. Кросскомпилятором можно считать и транслятор *О-Wintel-ОВМ* (рис. 3.15*в*).

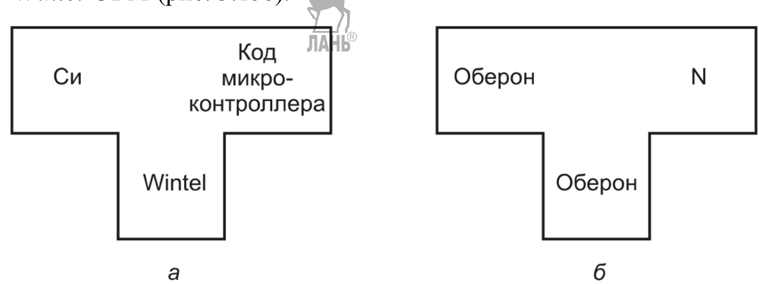


Рис. 3.16. Кросскомпилятор (*а*) и самокомпилятор (*б*)

[91] Таким способом, перечисляя фигурирующие на Т-диаграмме языки слева направо, условимся обозначать транслятор в тексте.

Самокомпилятор. Раскрутка

Компилятор может быть написан на своем собственном входном языке, в таком случае его называют самокомпилятором. Как это может быть реализовано и зачем это нужно?

Реализация старого языка для новой машины

Пусть, к примеру, имеется компилятор *Оберон-Оберон-N*, написанный на Обероне и транслирующий с Оберона в код машины[92] *N* (рис. 3.16*б*). Представим, что *N* — это новый тип компьютера, для которого еще нет ни одного компилятора никакого языка высокого уровня[93]. И мы решили, что первым языком на этой машине будет Оберон.

Оберон — не новый язык. Существует много компиляторов для него. Используем компилятор *Оберон-M-M*, имеющийся на машине *M* (рис. 3.17*а*). Собственно, с его помощью на машине *M* и велась разработка самокомпилятора. Транслируем самокомпилятор (рис. 3.17*б*). Получится кросскомпилятор *Оберон-M-N*, способный работать на машине *M* и транслирующий в код машины *N*. С его помощью исходный компилятор *Оберон-Оберон-N* транслируется еще раз (рис. 3.17*в*). В результате получается *Оберон-N-N*, способный работать на машине *N* и производить код для нее же. Получение нового компилятора двукратной трансляцией самокомпилятора может быть проиллюстрировано Т-диаграммами и немного по-другому (рис. 3.17*г*).

Реализация языка Оберон для новой машины выполнена. При этом не пришлось программировать в коде машины *N*.

Использование самокомпилятора облегчает перенос существующего языка на новую машину.

Если инструментальный компилятор *Оберон-M-M*, который был использован при разработке, доступен в исходном коде и тоже является самокомпилятором (*Оберон-Оберон-M*), то создание нового

---

[92] Будем для упрощения говорить о «машине» (вычислительной) вместо того, чтобы каждый раз использовать понятие программно-аппаратной платформы. В 1960-е, 1970-е и даже 1980-е годы компилятор действительно мог исполняться на «голом железе», не будучи погруженным в среду операционной системы. Сейчас такого уже нет.

[93] Конечно, новые типы машин появляются не так уж часто, но ведь появляются. К тому же, *N* может быть не реальным компьютером, а виртуальным.

транслятора Оберона могло сводиться всего лишь к модификации существующего — перепрограммированию его генератора кода.

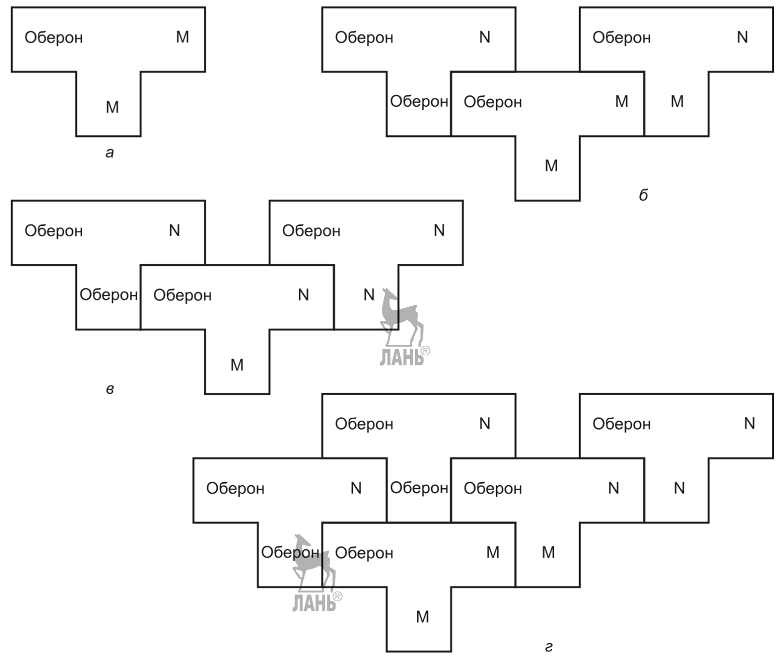


Рис. 3.17. Трансляция самокомпилятора: *а* — инструментальный компилятор; *б* — получение кросскомпилятора; *в* — самокомпиляция; *г* — полная схема раскрутки

Разработка компилятора нового языка раскруткой

Теперь рассмотрим другую ситуацию — создание компилятора нового языка, для которого никаких компиляторов еще не существует. Назовем новый язык *L*. Поставим задачу его реализации для машины *M*. Написать сразу самокомпилятор *L-L-M* возможно, скорее, теоретически. Да и как его использовать? Ведь компиляторов *L*, способных работать на машине *M*, нет, и компилировать этот компилятор нечем. В принципе, возможна его трансляция в ассемблерный код вручную,

но мы стремимся избежать программирования на языке низкого уровня.

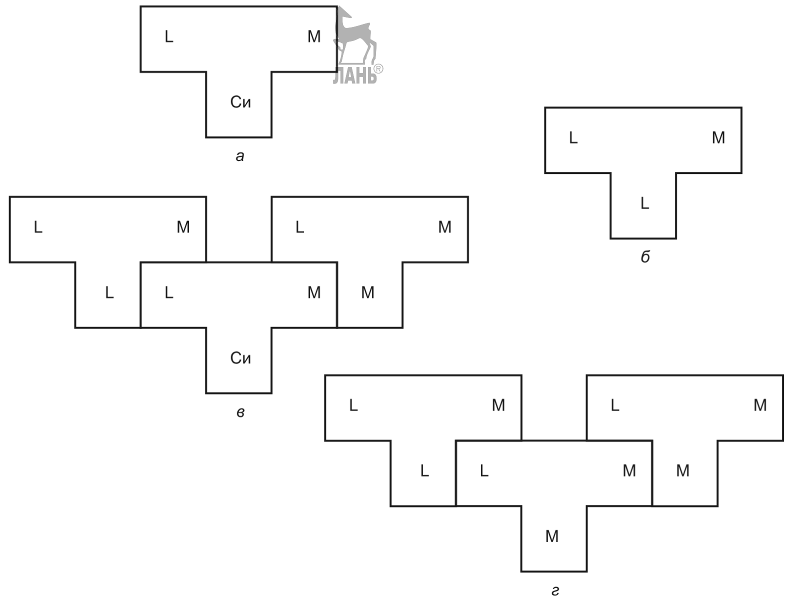


Рис. 3.18. Реализация нового языка: *а* — первоначальный компилятор; *б* — самокомпилятор; *в* — трансляция самокомпилятора; *г* — самокомпиляция

Поступим по-другому. Вначале запрограммируем компилятор *L* на каком-либо из существующих и реализованных для машины *M* языков, например, на Си (рис. 3.18*а*). После этого созданный компилятор можно переписать с языка Си на язык *L* (рис. 3.18*б*), используя в качестве инструмента при таком переносе первоначальный компилятор (рис 3.17*в*).

Самокомпилятор, существующий в виде программы в коде той машины, для которой он компилирует, при трансляции своего исходного текста порождает себя (рис. 3.18*г*).

Пользоваться языком *L* на машине *M* можно и, не имея самокомпилятора, достаточно применять *L-M-M*, полученный компиляцией *L-Си-M*. Такой транслятор даже может быть перепрограммированием

генератора кода перенесен на другую машину, где есть компилятор Си. Напомню, что самокомпилятор переносится на новую машину с такими же затратами, даже если там нет другого компилятора.

Другой способ получить самокомпилятор нового языка *L* для машины *M* состоит в следующем.

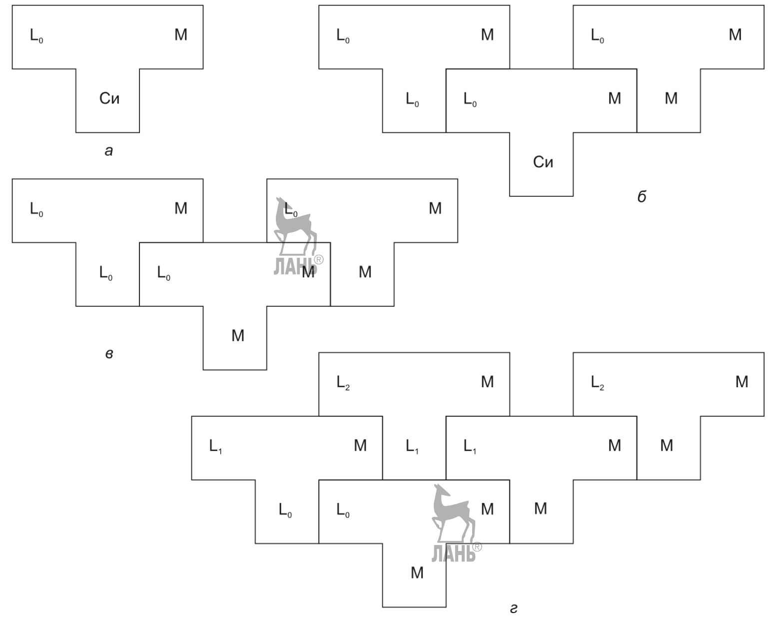


Рис. 3.19. Раскрутка

Вначале на каком-либо из существующих языков (например, Си) программируется компилятор для такого подмножества *L*, которое с одной стороны достаточно просто для того, чтобы не требовались большие затраты на его реализацию, с другой — достаточно богато, чтобы на этом языке можно было программировать компилятор. Обозначим такое подмножество L_0. Диаграмма начального компилятора L_0 показана на рисунке 3.19*а*.

Далее на языке L_0 программируется компилятор L_0, который компилируется с помощью L_0-*Си-М* (рис. 3.19*б*), после чего способен компилировать себя (рис. 3.19*в*). После этого начинается постепенное

расширение L_0 до тех пор, пока он не превратится в L. Сначала реализуется какое-либо из ранее не реализованных средств L. С помощью предыдущей версии компилируется новая, после чего вновь реализованное средство можно использовать в программе-компиляторе для его дальнейшего расширения (рис. 3.19*г*). Такой процесс называется *раскруткой*.

Собственно, частным случаем раскрутки является и процесс, рассмотренный в первой части этого раздела, когда на Си пишется компилятор сразу полного языка L. Это раскрутка, выполняемая за один шаг, когда $L_0 = L$.

Часто раскруткой называют вообще любой процесс создания самокомпилятора.

Улучшение качества кода при раскрутке

Самокомпилятор обладает полезным свойством — **способностью к самосовершенствованию.** Если он модернизируется с целью улучшения качества создаваемого кода, то через самокомпиляцию повышается и качество кода его самого.

Самокомпилятор является тестом для самого себя. Способность без ошибок компилировать такую нетривиальную программу, какой является транслятор, добавляет уверенности в надежности компилятора. Надо, однако, понимать, что самокомпилятор совсем не обязательно обеспечивает собственное полное тестовое покрытие[94].

Не могу удержаться еще от одного замечания. По моему убеждению, **только компилятор, написанный на своем собственном входном языке, может быть эстетически безупречен**. Если, конечно, речь идет о языке, который годится для написания компиляторов.

Примеры раскрутки

Известно достаточно много проектов, связанных с реализацией языков программирования, в ходе которых использовалась раскрутка.

Оптимизирующий компилятор Фортрана, известный как Fortran H, был написан на Фортране в конце 1960-х для операционной системы

[94] Тестовое покрытие характеризует, в какой степени различные части программы подвергаются проверке при тестировании. Считается, что минимальным требованием является исполнение каждого оператора тестируемой программы в ходе испытаний хотя бы один раз. В этом случае можно говорить о полном тестовом покрытии.

IBM OS/360. Раскрутка была выполнена в три стадии. Компилятор обеспечивал очень высокое качество объектного кода, но за счет использования четырех проходов и разнообразных оптимизаций сам работал медленно. Этот компилятор широко использовался в нашей стране в 1970–1980-е годы на ЕС ЭВМ в операционной системе ОС ЕС, которая была аналогом OS/360.

Первый компилятор Паскаля был написан в 1970 году на Паскале (Н. Вирт, У. Амман (U. Ammann), Е. Мармье (E. Marmier), Р. Шилд (R. Schild)). Затем он был вручную транслирован Р. Шилдом в код компьютера CDC6000. После чего прошел еще несколько стадий раскрутки.

С помощью раскрутки Н. Виртом и Ю. Гуткнехтом разрабатывались Оберон-система и компилятор Оберона. Первый вариант компилятора был получен из существующего на компьютере Lilith компилятора для Модулы-2. Это была смесь Модулы и Оберона, позволявшая обрабатывать подмножество Оберона (Оберон0). Затем компилятор Оберон0 был переведен на его собственный язык. Это позволило перенести его на компьютер Ceres, для которого транслятор и предназначался. Далее последовала длинная серия шагов раскрутки. Каждый шаг состоял из двух фаз: вначале в язык вносились изменения, затем эти изменения начинали использоваться в самом компиляторе.

Унификация промежуточного представления

Затраты на разработку трансляторов могут быть уменьшены, а мобильность повышена, если использовать унифицированное промежуточное представление программы при трансляции.

В роли такого представления может выступать язык программирования, язык некоторой (виртуальной) машины, различные формы внутреннего представления программы в компиляторе.

Промежуточные языки

Представим, что необходимо реализовать n языков для m машин. Если для каждой пары язык — машина создается компилятор, то всего потребуется $n \times m$ компиляторов. Можно использовать промежуточный язык. Обозначим его ПЯ. Для всех языков создается компилятор (конвертор), транслирующий в ПЯ. А для каждой машины — компилятор, транслирующий с ПЯ в код этой машины. Всего нужно разработать $n + m$ трансляторов. Даже, если речь идет о двух маши-

нах ($m = 2$), схема становится выгодна уже при $n = 3$. Промежуточный язык должен быть достаточно выразительным, чтобы транслировать на ПЯ было легко, и достаточно простым, чтобы оставались разумными затраты на создание трансляторов с ПЯ в машинный код.

Идея использования промежуточного языка возникла еще в конце 1950-х годов, когда обсуждался проект UNCOL (Universal Computer Oriented Language) — универсального компьютерного языка. Однако реального воплощения UNCOL тогда не получил.

Примером универсального промежуточного языка, специально созданного для этой роли, является АЛМО, разработанный в 1966 году в Институте прикладной математики АН СССР. Основанная на этом языке универсальная система программирования была реализована на нескольких типах отечественных ЭВМ.

На роль промежуточного языка хорошо подходит Си. Представляя собой, по выражению Н. Вирта, «синтаксически усовершенствованный ассемблер», он прост и эффективен. Компиляторы Си имеются практически на любой платформе. Язык хорошо стандартизован. И, если недостатком Си, как языка для программирования вручную, является низкая надежность, то при его использовании в роли промежуточного языка проблема исчезает. Конверторы, порождающие код на Си, неоднократно использовались при реализации разных языков, например, Си++.

Примером использования промежуточного языка при построении систем программирования является применение компанией Microsoft языка IL (от Intermediate Language — промежуточный язык) в многоязыковой системе .NET. IL представляет собой машинный код виртуальной стековой машины. Компиляторы языков Си#, Си++, Visual Basic, JScript, входящие в состав .NET, транслируют в IL. Затем IL-код преобразуется в машинный код платформы, на которой работает система. Такое преобразование может происходить непосредственно в момент загрузки IL-модуля. Основная форма существования IL-кода — двоичные файлы. Однако, существует и IL-ассемблер, который может быть использован для программирования на IL.

В принципе, в систему .NET могут быть добавлены и компиляторы других языков.

Заслуживает внимания технология, предложенная учеником Н. Вирта Михаэлем Францем (Michael Franz) в 1994 году. Вначале программа транслируется в промежуточное представление, которое

содержит компактную кодированную запись семантического дерева программы (SDE-представление). Кодирование основано на использовании так называемого семантического словаря (Semantic-Dictionary Encoding) и схоже с методами, используемыми при сжатии данных без потерь. SDE-представление полностью сохраняет информацию исходной программы, включая блочную структуру и сведения о типах выражений, облегчая тем самым последующую оптимизацию и генерацию кода.

Промежуточное представление программы преобразуется в код целевой платформы в момент загрузки, «на лету». Благодаря структуре SDE-представления кодогенерация может выполняться весьма эффективно с получением высококачественного кода. SDE-представление в два раза более компактно, чем машинный код традиционных процессоров и более компактно, чем байт-код Явы.

П-код и виртуальные машины

Другой способ использования промежуточного представления — трансляция с языка высокого уровня в код гипотетической (виртуальной) машины и последующая интерпретация полученного кода на целевой платформе с помощью программного симулятора. Такой подход нам хорошо знаком, именно он использован в этой книге.

Одним из первых примеров его применения стала разработка в 1973 году в Федеральном техническом университете (ETH) Цюриха группой Н. Вирта П-кода (в оригинале — P-code), П-машины и П-компилятора языка Паскаль. П-код представляет собой наилучшим образом приспособленный для трансляции с Паскаля машинный код гипотетической П-машины — компьютера со стековой архитектурой, реализованного с помощью интерпретатора, написанного на Паскале. Появление П-системы способствовало широкому распространению языка Паскаль, поскольку позволяло легко переносить его на различные платформы перепрограммированием П-машины.

Вариант П-кода долгое время использовался в системе программирования Visual Basic компании Microsoft.

В первой половине 1990-х идея П-кода была применена при реализации языка Ява компанией Sun Microsystems. Транслятор языка Ява преобразует программу в файлы байт-кода, который представляет собой машинный язык стекового компьютера. Байт-код исполняется интерпретатором, получившим название виртуальной Ява-машины (Java Virtual Machine, JVM).

Существуют реализации виртуальной Ява-машины для различных платформ. Все они могут исполнять одни и те же файлы байт-кода. Из-за того, что интерпретация существенно снижает быстродействие, разработаны различные схемы трансляции байт-кода в родной код той платформы, на которой исполняется программа. Часто такая трансляция происходит «на лету» при загрузке файлов байт-кода в память непосредственно перед исполнением. Выполняющие такую трансляцию подсистемы называют JIT-компиляторами (Just in time compilers, от Just in time — вовремя).

Идея использования промежуточного языка была реализована при создании отечественного многопроцессорного вычислительного комплекса «Эльбрус»[95], объектный код которого был фактически постфиксной записью. Во время выполнения программы на «Эльбрусе» происходила аппаратная «just-in-time» компиляция объектного кода в обычный трехадресный регистровый код.

За несколько лет до появления Ява-технологии использование виртуальной машины для унификации программного обеспечения было предложено в работах Ю. В. Матиясевича, А. Н. Терехова, Б. А. Федотова [Матиясевич, 1984], [Матиясевич, 1990].

Виртуальная машина, использованная в этой книге, безусловно, также происходит от П-кода[96]. Только в ней, в отличие от П-кода и JVM, реализована «чистая» стековая архитектура, когда операнды любых команд всегда находятся в стеке. Операнды команд П-кода и байт-кода Явы могут содержаться в самих командах.

Модульные компиляторы

Использование промежуточного представления в двух рассмотренных случаях (промежуточный язык и П-код) предполагает его формирование в явном виде с записью в файл. Кроме того, в процессе участвуют две программы. В первом варианте — это компилятор с исходного языка на промежуточный и компилятор с промежуточного языка в машинный код, во втором — компилятор в промежуточный код и интерпретатор этого кода.

---

[95] Идейным предшественником «Эльбруса» была американская вычислительная система Burroughs 6700/7700.

[96] Впервые эта машина под названием М-11 была применена в одном из учебных курсов в 1995 году, когда сведения о JVM были еще недоступны.

Использование и унификация промежуточного представления программы полезны и внутри компилятора или семейства компиляторов. При этом промежуточное представление может существовать лишь в виде некоторой внутренней структуры данных, а работа по трансляции выполняется единой программой. Пользователь компилятора может даже не знать о существовании промежуточного представления.

В компиляторе, построенном по модульному принципу, часть, зависящая от входного языка, отделяется от генерирующей части, которая зависит от целевой машины. Интерфейсом между этими частями является промежуточное представление. В компиляторе может присутствовать языково-независимый и машинно-независимый оптимизатор, который работает на уровне промежуточного представления, порождая его оптимизированный вариант (рис. 3.20).

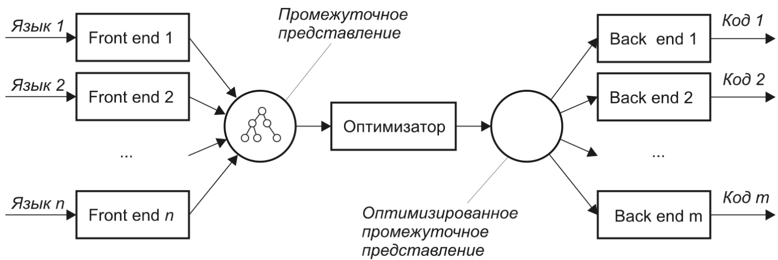


Рис. 3.20. Модульный компилятор

Фронтальная, анализирующая часть транслятора (front end) отвечает за лексический, синтаксический и контекстный анализ исходной программы и порождает промежуточное представление. При изменении входного языка фронтальная часть может быть заменена без того, чтобы это затронуло другие части транслятора.

Соответственно, синтезирующая часть (back end) не должна зависеть от входного языка и может подвергаться замене при изменении целевой машины.

Часто в роли промежуточного представления используется семантическое дерево программы. Хотя языки высокого уровня синтаксически различаются, семантическая основа широкого класса языков может быть систематизирована, а их промежуточное семантическое представление унифицировано.

Комбинирование различных анализирующих и синтезирующих частей позволяет получить семейство компиляторов для разных языков и целевых платформ примерно с теми же затратами, что и при использовании промежуточных языков. Однако модульный компилятор удобней в использовании, поскольку представляет собой единую программу.

В однопроходном же компиляторе, в частности том, который рассмотрен в этой книге, отвечающие за анализ и синтез части тесно переплетены. Эффективная и простая однопроходная схема не обладает той гибкостью в отношении входного языка и целевой машины, которая свойственна модульному компилятору.

По схеме с промежуточным представлением построены компиляторы XDS новосибирской компании Excelsior. На общей основе были разработаны оптимизирующие компиляторы для Модулы-2, Оберона, Явы, байт-кода Явы и двух специализированных языков, генераторы кода для процессоров Intel x86, Motorola 68K, SPARC, PowerPC, VAX, а также конверторы в Си и Си++. Модульную архитектуру имеют компиляторы Фортрана, Си, Си++ и Явы компании IBM.

Приложение. Язык программирования Оберон-2

Х. Мёссенбёк, Н. Вирт
Институт компьютерных систем, ETH Цюрих
Перевод с английского С. Свердлова

От переводчика

Язык программирования Оберон создан автором Паскаля и Модулы-2 Никлаусом Виртом в 1987 году в ходе разработки одноименной операционной системы для однопользовательской рабочей станции Ceres. Язык и операционная система названы именем одного из спутников планеты Уран — Оберона, открытого английским астрономом Уильямом Гершелем ровно за двести лет до описываемых событий.

«Сделай так просто, как возможно, но не проще того» — это высказывание А. Эйнштейна Вирт выбрал эпиграфом к описанию языка. Удивительно простой и даже аскетичный Оберон является, вероятно, минимальным универсальным языком высокого уровня. Он проще Паскаля и Модулы-2 и в то же время обогащает их рядом новых возможностей. Важно то, что автором языка руководили не сиюминутные коммерческие и конъюнктурные соображения, а глубокий анализ реальных программистских потребностей и стремление удовлетворить их простым, понятным, эффективным и безопасным образом, не вводя по возможности новых понятий. Являясь объектно-ориентированным языком, Оберон даже не содержит слова object.

Оберон представляется идеальным языком для изучения программирования. Сочетание простоты, строгости и неизбыточности предоставляет начинающему программисту великолепную возможность, не заблудившись в дебрях, выработать хороший стиль, освоив при этом и структурное и объектно-ориентированное и модульно-компонентное программирование.

Сотрудничество Н. Вирта с Ханспетером Мёссенбёком привело к добавлению в язык ряда новых средств. Новая версия получила название Оберон-2. Описание именно этого языка дается в настоящем переводе. Оберон-2 представляет собой почти правильное расширение Оберона. В Оберон-2 добавлены:

- связанные с типом процедуры;
- экспорт только для чтения;

- открытые массивы в качестве базового типа для указателей;
- оператор WITH с вариантами;
- оператор FOR.

Отдельного внимания заслуживает само описание, с которым вам предстоит познакомиться. Вирт и его соавтор достигли совершенства не только в искусстве разработки, но, несомненно, и в деле описания языков программирования. Поражают изумительная точность и краткость этого документа. Почти каждая его фраза превращается при написании компилятора в конкретные строки программного кода.

Возникшие при переводе описания Оберона-2 на русский язык терминологические вопросы решались исходя из следующих соображений: предпочтительным является буквальный перевод; недопустимо добавление терминов, отсутствующих в оригинале; должны быть учтены отечественные традиции в терминологии алголоподобных языков; предпочтительно использование терминов, привычных широкому кругу программистов, вместо узкоспециальных. Ниже приведен список терминов, перевод которых не представляется очевидным.

(direct) base type	(непосредственный) базовый тип
array compatible	совместимый массив
array type	тип массив
assignment compatible	совместимый по присваиванию
basic type	основной тип
browser	смотритель
case statement	оператор CASE
character	символ, знак
declaration	объявление
designator	обозначение
direct extension	непосредственное расширение
equal types	равные типы
exit statement	оператор выхода
expression compatible	совместимое выражение
for statement	оператор FOR
function procedure	процедура-функция
if statement	оператор IF
loop statement	оператор LOOP
matching	совпадение
operator	операция

pointer type	тип указатель
predeclared	стандартный
private field	скрытое поле
proper procedure	собственно процедура
public field	доступное поле
qualified	уточненный
real	вещественный
record type	тип запись
repeat statement	оператор REPEAT
return statement	оператор возврата
same type	одинаковый тип
scale factor	порядок
scope	область действия
statement	оператор
string	строка
symbol	слово
type extension	расширение типа
type guard	охрана типа
type inclusion	поглощение типа
type tag	тег
type test	проверка типа
type-bound procedures	связанные с типом процедуры
while statement	оператор WHILE
with statement	оператор WITH

1. Введение

Оберон-2 — язык программирования общего назначения, продолжающий традицию языков Паскаль и Модула-2. Его основные черты — блочная структура, модульность, раздельная компиляция, статическая типизация со строгим контролем соответствия типов (в том числе межмодульным), а также расширение типов и связанные с типами процедуры.

Расширение типов делает Оберон-2 объектно-ориентированным языком. Объект — это переменная абстрактного типа, содержащая данные (состояние объекта) и процедуры, которые оперируют этими данными. Абстрактные типы данных определены как расширяемые записи. Оберон-2 перекрывает большинство терминов объектно-ориентированных языков привычным словарем языков императивных, обходясь минимумом понятий в рамках тех же концепций.

Этот документ не является учебником программирования. Он преднамеренно краток. Его назначение — служить справочником для программистов, разработчиков компиляторов и авторов руководств. Если о чем-то не сказано, то обычно сознательно: или потому, что это следует из других правил языка, или потому, что потребовалось бы определять то, что фиксировать для общего случая представляется неразумным.

В приложении A определены некоторые термины, которые используются при описании правил соответствия типов Оберона-2. В тексте эти термины выделены курсивом, чтобы подчеркнуть их специальное значение (например, *одинаковый* тип).

2. Синтаксис

Для описания синтаксиса Оберона-2 используются Расширенные Бэкуса — Наура формы (РБНФ). Варианты разделяются знаком |. Квадратные скобки [и] означают необязательность записанного внутри них выражения, а фигурные скобки { и } означают его повторение (возможно 0 раз). Нетерминальные символы начинаются с заглавной буквы (например, Оператор). Терминальные символы или начинаются строчной буквой (например, идент), или записываются целиком заглавными буквами (например, BEGIN), или заключаются в кавычки (например, ":=").

3. Словарь и представление

Для представления терминальных символов предусматривается использование набора знаков ASCII. Слова языка — это идентификаторы, числа, строки, операции и разделители. Должны соблюдаться следующие лексические правила. Пробелы и концы строк не должны встречаться внутри слов (исключая комментарии и пробелы в символьных строках). Пробелы и концы строк игнорируются, если они не существенны для отделения двух последовательных слов. Заглавные и строчные буквы считаются различными.

1. *Идентификаторы* — последовательности букв и цифр. Первый символ должен быть буквой.

   идент = буква {буква | цифра}.

   Примеры: `x   Scan   Oberon2   GetSymbol   firstLetter`

2. *Числа* — целые или вещественные (без знака) константы. Типом целочисленной константы считается минимальный тип, которому принадлежит ее значение (см. п. 6.1). Если константа заканчивается буквой H, она является шестнадцатеричной, иначе — десятичной.
Вещественное число всегда содержит десятичную точку. Оно может также содержать десятичный порядок. Буква E (или D) означает «умножить на десять в степени». Вещественное число относится к типу REAL кроме случая, когда у него есть порядок, содержащий букву D. В этом случае оно относится к типу LONGREAL.

```
число = целое | вещественное.
целое = цифра {цифра} | цифра {шестнЦифра} "H".
вещественное = цифра {цифра} "." {цифра} [Порядок].
Порядок = ("E" | "D") ["+" | "-"] цифра {цифра}.
шестнЦифра = цифра | "A" | "B" | "C" | "D" | "E" | "F".
цифра = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Примеры:

`1991`	INTEGER	1991
`0DH`	SHORTINT	13
`12.3`	REAL	12.3
`4.567E8`	REAL	456 700 000
`0.57712566D-6`	LONGREAL	0.00000057712566

3. *Символьные* константы обозначаются порядковым номером символа в шестнадцатеричной записи, оканчивающейся буквой X.

```
символ = цифра {шестнЦифра} "X".
```

4. *Строки* — последовательности символов, заключенные в одиночные (`'`) или двойные (`"`) кавычки. Открывающая кавычка должна быть такой же, что и закрывающая и не должна встречаться внутри строки. Число символов в строке называется ее *длиной*. Строка длины 1 может использоваться везде, где допустима символьная константа и наоборот.

```
строка = ' " ' {символ} ' " ' | " ' " {символ} " ' ".
```

Примеры: `"Oberon-2"` `"Don't worry!"` `"x"`

5. *Операции* и *разделители* — это специальные символы, пары символов или зарезервированные слова, перечисленные ниже. Зарезерви-

рованные слова состоят исключительно из заглавных букв и не могут использоваться как идентификаторы.

```
+        :=       ARRAY        IMPORT        RETURN
-        ^        BEGIN        IN            THEN
*        =        BY           IS            TO
/        #        CASE         LOOP          TYPE
~        <        CONST        MOD           UNTIL
&        >        DIV          MODULE        VAR
.        <=       DO           NIL           WHILE
,        >=       ELSE         OF            WITH
;        ..       ELSIF        OR
|        :        END          POINTER
(        )        EXIT         PROCEDURE
[        ]        FOR          RECORD
{        }        IF           REPEAT
```

6. *Комментарии* могут быть вставлены между любыми двумя словами программы. Это произвольные последовательности символов, начинающиеся скобкой `(*` и оканчивающиеся `*)`. Комментарии могут быть вложенными. Они не влияют на смысл программы.

4. Объявления и области действия

Каждый идентификатор, встречающийся в программе, должен быть объявлен, если это не стандартный идентификатор. Объявления задают некоторые постоянные свойства объекта, например, является ли он константой, типом, переменной или процедурой. После объявления идентификатор используется для ссылки на соответствующий объект.

Область действия объекта *x* распространяется текстуально от точки его объявления до конца блока (модуля, процедуры или записи), в котором находится объявление. Для этого блока объект является *локальным*. Это разделяет области действия одинаково именованных объектов, которые объявлены во вложенных блоках. Правила для областей действия таковы:

1. Идентификатор не может обозначать больше чем один объект внутри данной области действия (то есть один и тот же идентификатор не может быть объявлен в блоке дважды);
2. Ссылаться на объект можно только изнутри его области действия;
3. Тип *T* вида POINTER TO *T1* (см. п. 6.4) может быть объявлен в точке, где *T1* еще неизвестен. Объявление *T1* должно следовать в том же блоке, в котором *T* является локальным;

4. Идентификаторы, обозначающие поля записи (см. п. 6.3) или процедуры, связанные с типом (см. п. 10.2), могут употребляться только в обозначениях записи.

Идентификатор, объявленный в блоке модуля, может сопровождаться при своем объявлении экспортной меткой («*» или «–»), чтобы указать, что он экспортируется. Идентификатор *x*, экспортируемый модулем *M*, может использоваться в других модулях, если они импортируют *M* (см. п. 11). Тогда идентификатор обозначается в этих модулях *M.x* и называется *уточненным идентификатором*. Переменные и поля записей, помеченные знаком «–» в их объявлении, предназначены *только для чтения* в модулях-импортерах.

```
УточнИдент = [идент "."] идент.
ИдентОпр   = идент ["*" | "–"].
```

Следующие идентификаторы являются стандартными; их значение определено в указанных разделах:

```
ABS       (10.3)   LEN       (10.3)
ASH       (10.3)   LONG      (10.3)
BOOLEAN   (6.1)    LONGINT   (6.1)
CAP       (10.3)   LONGREAL  (6.1)
CHAR      (6.1)    MAX       (10.3)
CHR       (10.3)   MIN       (10.3)
COPY      (10.3)   NEW       (10.3)
DEC       (10.3)   ODD       (10.3)
ENTIER    (10.3)   ORD       (10.3)
EXCL      (10.3)   REAL      (6.1)
FALSE     (6.1)    SET       (6.1)
HALT      (10.3)   SHORT     (10.3)
INC       (10.3)   SHORTINT  (6.1)
INCL      (10.3)   SIZE      (10.3)
INTEGER   (6.1)    TRUE      (6.1)
```

5. Объявления констант

Объявление константы связывает ее идентификатор с ее значением.

```
ОбъявлениеКонстанты= ИдентОпр "=" КонстантноеВыражение.
КонстантноеВыражение = Выражение.
```

Константное выражение — это выражение, которое может быть вычислено по его тексту без фактического выполнения программы. Его операнды — константы (п. 8) или стандартные функции (п. 10.3),

которые могут быть вычислены во время компиляции. Примеры объявлений констант:

```
N = 100
limit = 2*N - 1
fullSet = {MIN(SET)..MAX(SET)}
```

6. Объявления типов

Тип данных определяет набор значений, которые могут принимать переменные этого типа, и набор применимых операций. Объявление типа связывает идентификатор с типом. В случае структурированных типов (массивы и записи) объявление также определяет структуру переменных этого типа. Структурированный тип не может содержать сам себя.

```
ОбъявлениеТипа = ИдентОпр "=" Тип.
Тип = УточнИдент | ТипМассив | ТипЗапись | ТипУказатель |
  ПроцедурныйТип.
```

Примеры:

```
Table = ARRAY N OF REAL
Tree = POINTER TO Node
Node =  RECORD
   key: INTEGER;
   left, right: Tree
END
CenterTree = POINTER TO CenterNode
CenterNode = RECORD (Node)
   width: INTEGER;
   subnode: Tree
END
Function = PROCEDURE(x: INTEGER): INTEGER
```

6.1. Основные типы

Основные типы обозначаются стандартными идентификаторами. Соответствующие операции определены в п. 8.2, а стандартные функции в п. 10.3. Предусмотрены следующие основные типы:

1. BOOLEAN — логические значения TRUE и FALSE.
2. CHAR — символы расширенного набора ASCII (0X .. 0FFX).
3. SHORTINT — целые в интервале от MIN(SHORTINT) до MAX(SHORTINT).

4. INTEGER — целые в интервале от MIN(INTEGER) до MAX(INTEGER).
5. LONGINT — целые в интервале от MIN(LONGINT) до MAX(LONGINT).
6. REAL — вещественные числа в интервале от MIN(REAL) до MAX(REAL).
7. LONGREAL — вещественные числа от MIN(LONGREAL) до MAX(LONGREAL).
8. SET — множество из целых от 0 до MAX(SET).

Типы от 3 до 5 — *целые типы*, типы 6 и 7 — *вещественные типы*, а вместе они называются *числовыми типами*. Эти типы образуют иерархию; больший тип поглощает меньший тип:

LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT

6.2. Тип массив

Массив — структура, состоящая из определенного количества элементов одного типа, называемого *типом элементов*. Число элементов массива называется его *длиной*. Элементы массива обозначаются индексами, которые являются целыми числами от 0 до длины массива минус 1.

```
ТипМассив = ARRAY [Длина {"," Длина}] OF Тип.
Длина = КонстантноеВыражение.
```

Тип вида

```
ARRAY L0, L1, ..., Ln OF T
```

понимается как сокращение

```
ARRAY L0 OF
   ARRAY L1 OF
   …
      ARRAY Ln OF T
```

Массивы, объявленные без указания длины, называются *открытыми массивами*. Они могут использоваться только в качестве базового типа указателя (см. п. 6.4), типа элементов открытых массивов и типа формального параметра (см. п. 10.1). Примеры:

```
ARRAY 10, N OF INTEGER
ARRAY OF CHAR
```

6.3. Тип запись

Тип запись — структура, состоящая из фиксированного числа элементов, которые могут быть различных типов и называются *полями*. Объявление типа запись определяет имя и тип каждого поля. Область действия идентификаторов полей простирается от точки их объявления до конца объявления типа запись, но они также видимы внутри обозначений, ссылающихся на элементы переменных-записей (см. п. 8.1). Если тип запись экспортируется, то идентификаторы полей, которые должны быть видимы вне модуля, в котором объявлены, должны быть помечены. Они называются *доступными полями*; непомеченные элементы называются *скрытыми полями*.

```
ТипЗапись =
   RECORD ["(" БазовыйТип ")"]
   СписокПолей {";" СписокПолей} END.
БазовыйТип = УточнИдент.
СписокПолей = [СписокИдент ":" Тип].
```

Тип запись может быть объявлен как расширение другого типа запись. В примере

```
T0 = RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

T1 — (непосредственное) расширение *T0*, а *T0* — (непосредственный) базовый тип *T1* (см. прил. А). Расширенный тип *T1* состоит из полей своего базового типа и полей, которые объявлены в *T1*. Все идентификаторы, объявленные в расширенной записи, должны быть отличны от идентификаторов, объявленных в записи (записях) ее базового типа.

Примеры объявлений типа запись:

```
RECORD
   day, month, year: INTEGER
END
RECORD
   name, firstname: ARRAY 32 OF CHAR;
   age: INTEGER;
   salary: REAL
END
```

ЛАНЬ®

6.4. Тип указатель

Переменные-указатели типа *P* принимают в качестве значений указатели на переменные некоторого типа *T*. *T* называется базовым типом указателя типа *P* и должен быть типом массив или запись. Типы указатель заимствуют отношение расширения своих базовых типов: если тип *T1* — расширение *T* и *P1* — это тип POINTER TO *T1*, то *P1* — также является расширением *P*.

ТипУказатель = POINTER TO Тип.

Если *p* — переменная типа *P* = POINTER TO *T*, вызов стандартной процедуры *NEW(p)* (см. п. 10.3) размещает переменную типа *T* в свободной памяти. Если *T* — тип запись или тип массив с фиксированной длиной, размещение должно быть выполнено вызовом *NEW(p)*; если тип *T* — *n*-мерный открытый массив, размещение должно быть выполнено вызовом *NEW(p, e0, ..., en-1)*, чтобы *T* был размещен с длинами, заданными выражениями *e0, ..., en-1*. В любом случае указатель на размещенную переменную присваивается *p*. Переменная *p* имеет тип *P*. Переменная *p^*, на которую ссылается *p* (*динамическая переменная*), имеет тип *T*. Любая переменная-указатель может принимать значение NIL, которое не указывает ни на какую переменную вообще.

6.5. Процедурные типы

Переменные процедурного типа *T*, имеют значением процедуру (или NIL). Если процедура *P* присваивается переменной типа *T*, списки формальных параметров (см. п. 10.1) *P* и *T* должны *совпадать* (см. прил. A). *P* не должна быть стандартной или связанной с типом процедурой, и не может быть локальной в другой процедуре.

ПроцедурныйТип = PROCEDURE [ФормальныеПараметры].

7. Объявления переменных

Объявления переменных дают описание переменных, определяя идентификатор и тип данных для них.

ОбъявлениеПеременных = СписокИдент ":" Тип.

Переменные типа запись и указатель имеют как статический тип (тип, с которым они объявлены — называемый просто их типом), так и динамический тип (тип их значения во время выполнения). Для

указателей и параметров-переменных типа запись динамический тип может быть расширением их статического типа. Статический тип определяет какие поля записи доступны. Динамический тип используется для вызова связанных с типом процедур (см. п. 10.2).

Примеры объявлений переменных (со ссылками на примеры из п. 6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s: SET
F: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF RECORD
       name: ARRAY 32 OF CHAR;
       count: INTEGER
   END
t, c: Tree
```

8. Выражения

Выражения — конструкции, задающие правила вычисления по значениям констант и текущим значениям переменных других значений путем применения операций и процедур-функций. Выражения состоят из операндов и операций. Круглые скобки могут использоваться для группировки операций и операндов.

8.1. Операнды

За исключением конструкторов множества и литералов (чисел, символьных констант или строк), операнды представлены обозначениями. Обозначение содержит идентификатор константы, переменной или процедуры. Этот идентификатор может быть уточнен именем модуля (см. пп. 4 и 11) и может сопровождаться селекторами если обозначенный объект — элемент структуры.

```
Обозначение =
  УточнИдент {"." идент | "[" СписокВыражений "]"
  |"^" | "(" УточнИдент ")" }.
СписокВыражений = Выражение {"," Выражение}.
```

Если *a* — обозначение массива, *a[e]* означает элемент *a*, чей индекс — текущее значение выражения *e*. Тип *e* должен быть *целым* типом. Обозначение вида *a[e0, e1, ..., en]* применимо вместо *a[e0] [e1]*

... *[en]*. Если *r* обозначает запись, то *r.f* означает поле *f* записи *r* или процедуру *f*, связанную с динамическим типом *r* (п. 10.2). Если *p* обозначает указатель, *p^* означает переменную, на которую ссылается *p*. Обозначения *p^.f* и *p^[e]* могут быть сокращены до *p.f* и *p[e]*, то есть запись и индекс массива подразумевают разыменование. Если *a* или *r* доступны только для чтения, то *a[e]* и *r.f* также предназначены только для чтения.

Охрана типа v(T) требует, чтобы динамическим типом *v* был *T* (или расширение *T*), то есть выполнение программы прерывается, если динамический тип *v* — не *T* (или расширение *T*). В пределах такого обозначения *v* воспринимается как имеющая статический тип *T*. Охрана применима, если

1. *v* — параметр-переменная типа запись, или *v* — указатель, и если
2. *T* — расширение статического типа *v*.

Если обозначенный объект — константа или переменная, то обозначение ссылается на их текущее значение. Если он — процедура, то обозначение ссылается на эту процедуру, если только обозначение не сопровождается (возможно пустым) списком параметров. В последнем случае подразумевается активация процедуры и подстановка значения результата, полученного при ее исполнении. Фактические параметры должны соответствовать формальным параметрам как и при вызовах собственно процедуры (см. п. 10.1).

Примеры обозначений (со ссылками на примеры из п. 7):

`i`	(INTEGER)
`a[i]`	(REAL)
`w[3].name[i]`	(CHAR)
`t.left.right`	(Tree)
`t(CenterTree).subnode`	(Tree)

8.2. Операции

В выражениях синтаксически различаются четыре класса операций с разными приоритетами (порядком выполнения). Операция ~ имеет самый высокий приоритет, далее следуют операции типа умножения, операции типа сложения и отношения. Операции одного приоритета выполняются слева направо. Например, x–y–z означает (x–y) –z.

Выражение = ПростоеВыражение [Отношение ПростоеВыражение].
ПростоеВыражение = ["+" | "-"] Слагаемое
{ОперацияСложения Слагаемое}.

Слагаемое = Множитель {ОперацияУмножения Множитель}.
Множитель =
Обозначение [ФактическиеПараметры] |число | символ |
строка | NIL | Множество | "(" Выражение ")" | "~" Множитель.
Множество = "{" [Элемент {"," Элемент}] "}".
Элемент = Выражение [".." Выражение].
ФактическиеПараметры = "(" [СписокВыражений] ")".
Отношение = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
ОперацияСложения = "+" | "-" | OR.
ОперацияУмножения = "*" | "/" | DIV | MOD | "&".

Предусмотренные операции перечислены в следующих таблицах. Некоторые операции применимы к операндам различных типов, обозначая разные действия. В этих случаях фактическая операция определяется типом операндов. Операнды должны быть *совместимыми выражениями* для данной операции (см. прил. А).

8.2.1. Логические операции

OR	Логическая дизъюнкция	P OR q	«если p, то TRUE, иначе q»
&	Логическая конъюнкция	P & q	«если p то q, иначе FALSE»
~	Отрицание	~p	«не p»

Эти операции применимы к операндам типа BOOLEAN и дают результат типа BOOLEAN.

8.2.2. Арифметические операции

+	сумма
–	разность
*	произведение
/	вещественное деление
DIV	деление нацело
MOD	остаток

Операции +, –, *, и / применимы к операндам *числовых типов*. Тип их результата — тип того операнда, который поглощает тип другого операнда, кроме деления (/), чей результат — наименьший *вещественный тип*, который поглощает типы обоих операндов. При использовании в качестве одноместной операции "–" обозначает пере-

мену знака, а "+" — тождественную операцию. Операции DIV и MOD применимы только к целочисленным операндам. Они связаны следующими формулами, определенными для любого *x* и положительного делителя *y*:

x = (x DIV y) * y + (x MOD y)
0 <= (x MOD y) < y

Примеры:

x	y	x DIV y	x MOD y
5	3	1	2
–5	3	–2	1

8.2.3. Операции над множествами

+ объединение
– разность (x – y = x * (–y))
* пересечение
/ симметрическая разность множеств (x / y = (x–y) + (y–x))

Эти операции применимы к операндам типа SET и дают результат типа SET. Одноместный «минус» обозначает дополнение *x*, то есть –*x* — это множество целых между 0 и MAX(SET), которые не являются элементами *x*. Операции с множествами не ассоциативны ((*a*+*b*)–*c* # *a*+(*b*–*c*)).

Конструктор множества задает значение множества списком элементов, заключенным в фигурные скобки. Элементы должны быть целыми в диапазоне 0 ... MAX(SET). Диапазон *a*...*b* обозначает все целые числа в интервале [*a, b*].

8.2.4. Отношения

= равно
не равно
< меньше
<= меньшее или равно
> больше
>= больше или равно
IN принадлежность множеству
IS проверка типа

Отношения дают результат типа BOOLEAN. Отношения =, #, <, <=, > и >= применимы к *числовым типам*, типу CHAR, строкам и символьным массивам, содержащим в конце 0X. Отношения = и # кроме

того применимы к типам BOOLEAN и SET, а также к указателям и процедурным типам (включая значение NIL). *x IN s* означает «*x* является элементом *s*». *x* должен быть *целого типа*, а *s* — типа SET. *v IS T* означает «динамический тип *v* есть *T* (или расширение *T*)» и называется проверкой типа. Проверка типа применима, если
1. *v* — параметр-переменная типа запись, или *v* — указатель, и если
2. *T* — расширение статического типа *v*.

Примеры выражений (со ссылками на примеры из п. 7):

`1991`	INTEGER
`i DIV 3`	INTEGER
`~p OR q`	BOOLEAN
`(i+j) * (i-j)`	INTEGER
`s - {8, 9, 13}`	SET
`i + x`	REAL
`a[i+j] * a[i-j]`	REAL
`(0<=i) & (i<100)`	BOOLEAN
`t.key = 0`	BOOLEAN
`k IN {i..j-1}`	BOOLEAN
`w[i].name <= "John"`	BOOLEAN
`t IS CenterTree`	BOOLEAN

9. Операторы

Операторы обозначают действия. Есть простые и структурные операторы. Простые операторы не содержат в себе никаких частей, которые являются самостоятельными операторами. Простые операторы — присваивание, вызов процедуры, операторы возврата и выхода. Структурные операторы состоят из частей, которые являются самостоятельными операторами. Они используются, чтобы выразить последовательное и условное, выборочное и повторное исполнение. Оператор также может быть пустым, в этом случае он не обозначает никакого действия. Пустой оператор добавлен, чтобы упростить правила пунктуации в последовательности операторов.

Оператор =
[Присваивание | ВызовПроцедуры | ОператорIF | ОператорCASE | ОператорWHILE | ОператорREPEAT | ОператорFOR | ОператорLOOP | ОператорWITH | EXIT | RETURN [Выражение]].

9.1. Присваивания

Присваивание заменяет текущее значение переменной новым значением, определяемым выражением. Выражение должно быть *совместимо по присваиванию* с переменной (см. Приложение. А). Знаком операции присваивания является ":=", который читается “присвоить”.

Присваивание = Обозначение ":=" Выражение.

Если выражение *e* типа *Te* присваивается переменной *v* типа, *Tv*, имеет место следующее:

Если *Tv* и *Te* — записи, то в присваивании участвуют только те поля *Te*, которые также имеются в *Tv* (проецирование); динамический тип *v* и статический тип *v* должны быть *одинаковы*, и не изменяются присваиванием;

Если *Tv* и *Te* — типы указатель, динамическим типом *v* становится динамический тип *e*;

Если *Tv* это ARRAY *n* OF CHAR, а *e* — строка длины $m < n$, $v[i]$ присваиваются значения *ei* для $i = 0 \dots m\text{-}1$, а $v[m]$ получает значение 0X.

Примеры присваиваний (со ссылками на примеры из п. 7):

```
i := 0
p := i = j
x := i + 1
k := log2(i+j)
F := log2            (* см. 10.1 *)
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y)*(x-y)
t.key := i
w[i+1].name := "John"
t := c
```

9.2. Вызовы процедур

Вызов процедуры активирует процедуру. Он может содержать список фактических параметров, которые заменяют соответствующие формальные параметры, определенные в объявлении процедуры (см. п. 10). Соответствие устанавливается в порядке следования параметров в списках фактических и формальных параметров. Имеются два вида параметров: параметры-переменные и параметры-значения.

Если формальный параметр — параметр-переменная, соответствующий фактический параметр должен быть обозначением перемен-

ной. Если фактический параметр обозначает элемент структурной переменной, селекторы компонент вычисляются, когда происходит замена формальных параметров фактическими, то есть перед выполнением процедуры. Если формальный параметр — параметр-значение, соответствующий фактический параметр должен быть выражением. Это выражение вычисляется перед вызовом процедуры, а полученное в результате значение присваивается формальному параметру (см. также п. 10.1).

ВызовПроцедуры = Обозначение [ФактическиеПараметры].

Примеры:

```
WriteInt(i*2+1)     (* см. 10.1 *)
INC(w[k].count)
t.Insert("John")    (* см. 11 *)
```

9.3. Последовательность операторов

Последовательность операторов, разделенных точкой с запятой, означает поочередное выполнение действий, заданных составляющими операторами.

ПоследовательностьОператоров = Оператор {";" Оператор}.

9.4. Операторы IF

ОператорIF =
  IF Выражение THEN ПоследовательностьОператоров
  {ELSIF Выражение THEN ПоследовательностьОператоров}
  [ELSE ПоследовательностьОператоров]
  END.

Операторы IF задают условное выполнение входящих в них последовательностей операторов. Логическое выражение, предшествующие последовательности операторов, будем называть условием[97]. Условия проверяются последовательно одно за другим, пока очередное не окажется равным TRUE, после чего выполняется связанная с этим условием последовательность операторов. Если ни одно условие не удовлетворено, выполняется последовательность операторов, записанная после слова ELSE, если оно имеется.

[97] В оригинале — guard. *Прим. перев.*

Пример:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF (ch = "'") OR (ch = '"') THEN ReadString
ELSE SpecialCharacter
END
```

9.5. Операторы CASE

Операторы CASE определяют выбор и выполнение последовательности операторов по значению выражения. Сначала вычисляется выбирающее выражение, а затем выполняется та последовательность операторов, чей список меток варианта содержит полученное значение. Выбирающее выражение должно быть такого *целого типа*, который включает типы всех меток вариантов, или и выбирающее выражение и метки вариантов должны иметь тип CHAR. Метки варианта — константы, и ни одно из их значений не должно употребляться больше одного раза. Если значение выражения не совпадает с меткой ни одного из вариантов, выбирается последовательность операторов после слова ELSE, если оно есть, иначе программа прерывается.

ОператорCASE =
CASE Выражение OF Вариант {" | " Вариант}
[ELSE ПоследовательностьОператоров] END.
Вариант = [СписокМетокВарианта":"ПоследовательностьОператоров].
СписокМетокВарианта = МеткиВарианта {"," МеткиВарианта }.
МеткиВарианта=КонстантноеВыражение[".."КонстантноеВыражение].

Пример:

```
CASE ch OF
   "A".."Z": ReadIdentifier
|  "0".."9": ReadNumber
|  "'", '"': ReadString
ELSE SpecialCharacter
END
```

9.6. Операторы WHILE

Операторы WHILE задают повторное выполнение последовательности операторов, пока логическое выражение (условие) остается равным TRUE. Условие проверяется перед каждым выполнением последовательности операторов.

```
ОператорWHILE =
  WHILE Выражение DO
    ПоследовательностьОператоров
  END.
```

Примеры:

```
WHILE i > 0 DO i := i DIV 2; k := k + 1 END
WHILE (t # NIL) & (t.key # i) DO t := t.left END
```

9.7. Операторы REPEAT

Оператор REPEAT определяет повторное выполнение последовательности операторов пока условие, заданное логическим выражением, не удовлетворено. Последовательность операторов выполняется по крайней мере один раз.

```
ОператорREPEAT =
  REPEAT ПоследовательностьОператоров UNTIL Выражение.
```

9.8. Операторы FOR

Оператор FOR определяет повторное выполнение последовательности операторов фиксированное число раз для прогрессии значений целочисленной переменной, называемой *управляющей переменной* оператора FOR.

```
ОператорFOR =
  FOR идент ":=" Выражение TO Выражение
  [BY КонстантноеВыражение]
  DO ПоследовательностьОператоров END.
```

Оператор

```
FOR v := beg TO end BY step DO statements END
```

эквивалентен

```
temp := end; v := beg;
IF step > 0 THEN
   WHILE v <= temp DO statements; v := v + step END
ELSE
   WHILE v >= temp DO statements; v := v + step END
END
```

temp и *v* имеют *одинаковый* тип. Шаг (*step*) должен быть отличным от нуля константным выражением. Если шаг не определен, он принимается равным 1.

Примеры:

```
FOR i := 0 TO 79 DO k := k + a[i] END
FOR i := 79 TO 1 BY -1 DO a[i] := a[i-1] END
```

9.9. Операторы LOOP

Оператор LOOP определяет повторное выполнение последовательности операторов. Он завершается после выполнения оператора выхода внутри этой последовательности (см. п. 9.10).

ОператорLOOP = LOOP ПоследовательностьОператоров END.

Пример:

```
LOOP
   ReadInt(i);
   IF i < 0 THEN EXIT END;
   WriteInt(i)
END
```

Операторы LOOP полезны, чтобы выразить повторения с несколькими точками выхода, или в случаях, когда условие выхода находится в середине повторяемой последовательности операторов.

9.10. Операторы возврата и выхода

Оператор возврата выполняет завершение процедуры. Он обозначается словом RETURN, за которым следует выражение, если процедура является процедурой-функцией. Тип выражения должен быть *совместим по присваиванию* (см. прил. А) с типом результата, определенным в заголовке процедуры (см. п. 10).

Процедуры-функции должны быть завершены оператором возврата, задающим значение результата. В собственно процедурах оператор возврата подразумевается в конце тела процедуры. Любой явный оператор появляется, следовательно, как дополнительная (вероятно, для исключительной ситуации) точка завершения.

Оператор выхода обозначается словом EXIT. Он определяет завершение охватывающего оператора LOOP и продолжение выполнения программы с оператора, следующего за оператором LOOP. Оператор выхода связан с содержащим его оператором цикла контекстуально, а не синтаксически.

9.11. Операторы WITH

Операторы WITH выполняют последовательность операторов в зависимости от результата проверки типа и применяют охрану типа к каждому вхождению проверяемой переменной внутри этой последовательности операторов.

```
ОператорWITH =
  WITH Охрана DO ПоследовательностьОператоров
  {"|" Охрана DO ПоследовательностьОператоров}
  [ELSE ПоследовательностьОператоров] END.
Охрана = УточнИдент ":" УточнИдент.
```

Если *v* — параметр-переменная типа запись или переменная-указатель, и если ее статический тип *T0*, оператор

```
WITH v: T1 DO S1 | v: T2 DO S2 ELSE S3 END
```

имеет следующий смысл: если динамический тип *v* — *T1*, то выполняется последовательность операторов *S1* в которой *v* воспринимается так, будто она имеет статический тип *T1*; иначе, если динамический тип *v* — *T2*, выполняется *S2*, где *v* воспринимается как имеющая статический тип *T2*; иначе выполняется *S3*. *T1* и *T2* должны быть расширениями *T0*. Если ни одна проверка типа не удовлетворена, а ELSE отсутствует, программа прерывается.

Пример:

```
WITH t: CenterTree DO i := t.width; c := t.subnode END
```

10. Объявления процедур

Объявление процедуры состоит из заголовка процедуры и тела процедуры. Заголовок определяет имя процедуры и формальные параметры. Для связанных с типом процедур в объявлении также определяется параметр-приемник. Тело содержит объявления и операторы. Имя процедуры повторяется в конце объявления процедуры.

Имеются два вида процедур: собственно процедуры и процедуры-функции. Последние активизируются обозначением функции как часть выражения и возвращают результат, который является операндом выражения. Собственно процедуры активизируются вызовом процедуры. Процедура является процедурой-функцией, если ее формальные параметры задают тип результата. Тело процедуры-функции должно содержать оператор возврата, который определяет результат.

Все константы, переменные, типы и процедуры, объявленные внутри тела процедуры, локальны в процедуре. Поскольку процедуры тоже могут быть объявлены как локальные объекты, объявления процедур могут быть вложенными. Вызов процедуры изнутри ее объявления подразумевает рекурсивную активацию.

Объекты, объявленные в окружении процедуры, также видимы в тех частях процедуры, в которых они не перекрыты локально объявленным объектом с тем же самым именем.

```
ОбъявлениеПроцедуры =
  ЗаголовокПроцедуры ";" ТелоПроцедуры идент.
ЗаголовокПроцедуры =
  PROCEDURE [Приемник] ИдентОпр [ФормальныеПараметры].
ТелоПроцедуры =
  ПоследовательностьОбъявлений
  [BEGIN ПоследовательностьОператоров] END.
ПослОбъявлений =
  {CONST {ОбъявлениеКонстант ";"} |
  TYPE{ОбъявлениеТипов ";"}
  | VAR {ОбъявлениеПеременных ";"}}
  {ОбъявлениеПроцедуры ";" | ОпережающееОбъявление";"}.
ОпережающееОбъявление =
  PROCEDURE "^" [Приемник] ИдентОпр [ФормальныеПараметры].
```

Если объявление процедуры содержит параметр-приемник, процедура рассматривается как связанная с типом (см. п. 10.2). Опережающее объявление служит, чтобы разрешить ссылки на процедуру, чье фактическое объявление появляется в тексте позже. Списки формальных параметров опережающего объявления и фактического объявления должны *совпадать* (см. прил. А).

10.1. Формальные параметры

Формальные параметры — идентификаторы, объявленные в списке формальных параметров процедуры. Им соответствуют фактические параметры, которые задаются при вызове процедуры. Подстановка фактических параметров вместо формальных происходит при вызове процедуры. Имеются два вида параметров: *параметры-значения* и *параметры-переменные*, обозначаемые в списке формальных параметров отсутствием или наличием ключевого слова VAR. Парамет-

ры-значения — это локальные переменные, которым в качестве начального присваивается значение соответствующего фактического параметра. Параметры-переменные соответствуют фактическим параметрам, которые являются переменными, и означают эти переменные. Область действия формального параметра простирается от его объявления до конца блока процедуры, в котором он объявлен. Процедура-функция без параметров должна иметь пустой список параметров. Она должна вызываться обозначением функции, чей список фактических параметров также пуст. Тип результата процедуры не может быть ни записью, ни массивом.

```
ФормальныеПараметры =
   "(" [СекцияФП {";" СекцияФП }] ")" [":" УточненИдент].
СекцияФП =
   [VAR] идент {"," идент} ":" Тип.
```

Пусть *Tf* — тип формального параметра *f* (не открытого массива) и *Ta* — тип соответствующего фактического параметра *a*. Для параметров-переменных *Ta* и *Tf* должны быть *одинаковыми* типами или *Tf* должен быть типом запись, а *Ta* — расширением *Tf*. Для параметров-значений *a* должен быть *совместим по присваиванию* с *f* (см. прил. А).

Если *Tf* — открытый массив, то *a* должен быть *совместимым массивом* для *f* (см. прил. А). Длина *f* становится равной длине *a*.

Примеры объявлений процедур:

```
PROCEDURE ReadInt (VAR x: INTEGER);
   VAR i: INTEGER; ch: CHAR;
BEGIN
   i := 0; Read(ch);
   WHILE ("0" <= ch) & (ch <= "9") DO
      i := 10*i + (ORD(ch)-ORD("0")); Read(ch)
   END;
   x := i
END ReadInt

PROCEDURE WriteInt (x: INTEGER); (* 0<=x<100000 *)
   VAR i: INTEGER; buf: ARRAY 5 OF INTEGER;
BEGIN
   i := 0;
   REPEAT buf[i] := x MOD 10; x := x DIV 10; INC(i)
   UNTIL x = 0;
   REPEAT DEC(i); Write(CHR(buf[i] + ORD("0")))
   UNTIL i = 0
```

```
END WriteInt

PROCEDURE WriteString (s: ARRAY OF CHAR);
   VAR i: INTEGER;
BEGIN i := 0;
   WHILE (i < LEN(s)) & (s[i] # 0X) DO
      Write(s[i]); INC(i)
   END
END WriteString;

PROCEDURE log2(x: INTEGER): INTEGER;
   VAR y: INTEGER;   (*предполагается x>0*)
BEGIN
   y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
   RETURN y
END log2
```

10.2. Процедуры, связанные с типом

Глобально объявленные процедуры могут быть ассоциированы с типом запись, объявленным в том же самом модуле. В этом случае говорится, что процедуры *связаны* с типом запись[98]. Связь выражается типом приемника в заголовке объявления процедуры. Приемник может быть или параметром-переменной типа *T*, если *T* — тип запись, или параметром-значением типа POINTER TO *T* (где *T* — тип запись). Процедура, связанная с типом *T*, рассматривается как локальная для него.

```
ЗаголовокПроцедуры =
   PROCEDURE [Приемник] ИдентОпр [ФормальныеПараметры].
Приемник = "(" [VAR] имя ":" имя ")".
```

Если процедура *P* связана с типом *T0*, она неявно также связана с любым типом *T1*, который является расширением *T0*. Однако процедура *P′* (с тем же самым именем, что и *P*) может быть явно связана с *T1,* перекрывая в этом случае связывание с *P*. *P′* рассматривается как переопределение *P* для *T1*. Формальные параметры *P* и *P′* должны *совпадать* (см. прил. А).

Если *P* и *T1* экспортируются (см. п. 4), *P′* также должна экспортироваться.

[98] Будем называть их также «связанные процедуры». *Прим. перев.*

Если v — обозначение, а P — связанная процедура, то $v.P$ обозначает процедуру P, связанную с динамическим типом v. Заметим, что это может быть процедура, отличная от той, что связана со статическим типом v. v передается приемнику процедуры P согласно правилам передачи параметров, определенным в п. 10.1.

Если r — параметр-приемник, объявленный с типом T, $r.P$^ обозначает (переопределенную) процедуру P, связанную с базовым для T типом. В опережающем объявлении связанной процедуры и в фактическом объявлении процедуры параметр-приемник должен иметь *одинаковый* тип. Списки формальных параметров в обоих объявлениях должны совпадать (см. прил. А).

Примеры:

```
PROCEDURE (t: Tree) Insert (node: Tree);
   VAR p, father: Tree;
BEGIN p := t;
   REPEAT father := p;
      IF node.key = p.key THEN RETURN END;
      IF node.key < p.key THEN p := p.left
      ELSE p := p.right END
   UNTIL p = NIL;
   IF node.key < father.key THEN father.left := node
   ELSE father.right := node END;
   node.left := NIL; node.right := NIL
END Insert;

PROCEDURE (t: CenterTree) Insert (node: Tree);
(*переопределение*)
BEGIN
   WriteInt(node(CenterTree).width);
   t.Insert^(node)
      (* вызывает процедуру Insert, связанную с Tree
*)
END Insert;
```

10.3. Стандартные процедуры

Следующая таблица содержит список стандартных процедур. Некоторые процедуры — обобщенные, то есть они применимы к операндам нескольких типов. Буква v обозначает переменную, x и n — выражения, T — тип.

Процедуры-функции

Название	Тип аргумента	Тип результата	Функция
ABS(x)	числовой тип	совпадает с типом x	абсолютное значение
ASH(x, n)	x, n: целый тип	LONGINT	арифметический сдвиг ($x*2^n$)
CAP(x)	CHAR	CHAR	x — буква: соответствующая заглавная буква
CHR(x)	целый тип	CHAR	символ с порядковым номером x
ENTIER(x)	вещественный тип	LONGINT	наибольшее целое, не превосходящее x
LEN(v, n)	v: массив; n: целая константа	LONGINT	длина v в измерении n (первое измерение = 0)
LEN(v)	v: массив	LONGINT	равносильно LEN(v, 0)
LONG(x)	SHORTINT	INTEGER	тождество
	INTEGER	LONGINT	
	REAL	LONGREAL	
MAX(T)	T = основной тип	T	наибольшее значение типа T
	T = SET	INTEGER	наибольший элемент множества
MIN(T)	T = основной тип	T	наименьшее значение типа T
	T = SET	INTEGER	0
ODD(x)	целый тип	BOOLEAN	x MOD 2 = 1
ORD(x)	CHAR	INTEGER	порядковый номер x
SHORT(x)	LONGINT	INTEGER	тождество
	INTEGER	SHORTINT	тождество
	LONGREAL	REAL	тождество (возможно усечение)
SIZE(T)	любой тип	целый тип	число байт, занимаемых T

Собственно процедуры

Название	Типы аргументов	Функция
ASSERT(x)	x: логическое выражение	прерывает выполнение программы, если не x
ASSERT(x, n)	x: логическое выражение; n: целая константа	прерывает выполнение программы, если не x
COPY(x, v)	x: символьный массив, строка; v: символьный массив	v := x
DEC(v)	целый тип	v := v – 1
DEC(v, n)	v, n: целый тип	v := v – n
EXCL(v, x)	v: SET; x: целый тип	v := v – {x}
HALT(n)	целая константа	прерывает выполнение программы
INC(v)	целый тип	v := v + 1
INC(v, n)	v, n: целый тип	v := v + n
INCL(v, x)	v: SET; x: целый тип	v := v + {x}
NEW(v)	указатель на запись или массив фиксированной длины	размещает v^
NEW(v, x0, ..., xn)	v: указатель на открытый массив; xi: целый тип	размещает v^ с длинами x0... xn

COPY разрешает присваивание строки или символьного массива, содержащего ограничитель 0X, другому символьному массиву. В случае необходимости, присвоенное значение усекается до длины получателя минус один. Получатель всегда будет содержать 0X как ограничитель. В ASSERT(x, n) и HALT(n), интерпретация *n* зависит от реализации основной системы.

11. Модули

Модуль — совокупность объявлений констант, типов, переменных и процедур вместе с последовательностью операторов, предназначенных для присваивания начальных значений переменным. Модуль представляет собой текст, который является единицей компиляции.

```
Модуль =
   MODULE идент ";" [СписокИмпорта]
   ПоследовательностьОбъявлений
```

[BEGIN ПоследовательностьОператоров] END идент ".".
СписокИмпорта = IMPORT Импорт {"," Импорт} ";".
Импорт = [идент ":="] идент.

Список импорта определяет имена импортируемых модулей. Если модуль *A* импортируется модулем *M*, и *A* экспортирует идентификатор *x*, то *x* упоминается внутри *M* как *A.x*. Если *A* импортируется как *B:=A*, объект *x* должен вызываться как *B.x*. Это позволяет использовать короткие имена-псевдонимы в уточненных идентификаторах. Модуль не должен импортировать себя. Идентификаторы, которые экспортируются (то есть должны быть видимы в модулях-импортерах) нужно отметить экспортной меткой в их объявлении (см. п. 4).

Последовательность операторов после символа BEGIN выполняется, когда модуль добавляется к системе (загружается). Это происходит после загрузки импортируемых модулей. Отсюда следует, что циклический импорт модулей запрещен. Отдельные (не имеющие параметров и экспортированные) процедуры могут быть активированы из системы. Эти процедуры служат командами (см. прил. D1).

```
MODULE Trees;
 (* экспорт: Tree, Node, Insert, Search, Write, Init *)
 (* экспорт только для чтения: Node.name *)
   IMPORT Texts, Oberon;
   TYPE
      Tree* = POINTER TO Node;
      Node* = RECORD
         name-: POINTER TO ARRAY OF CHAR;
         left, right: Tree
      END;
   VAR w: Texts.Writer;

   PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
      VAR p, father: Tree;
   BEGIN p := t;
      REPEAT father := p;
         IF name = p.name^ THEN RETURN END;
         IF name < p.name^ THEN p := p.left
         ELSE p := p.right END
      UNTIL p = NIL;
      NEW(p); p.left := NIL; p.right := NIL;
      NEW(p.name, LEN(name)+1);
      COPY(name, p.name^);
```

```
      IF name < father.name^ THEN father.left := p
      ELSE father.right := p END
   END Insert;

   PROCEDURE(t:Tree)Search*(name:ARRAY OF CHAR):Tree;
      VAR p: Tree;
   BEGIN p := t;
      WHILE (p # NIL) & (name # p.name^) DO
         IF name < p.name^ THEN p := p.left
         ELSE p := p.right END
      END;
      RETURN p
   END Search;

   PROCEDURE (t: Tree) Write*;
   BEGIN
      IF t.left # NIL THEN t.left.Write END;
      Texts.WriteString(w, t.name^); Texts.WriteLn(w);
      Texts.Append(Oberon.Log, w.buf);
      IF t.right # NIL THEN t.right.Write END
   END Write;

   PROCEDURE Init* (t: Tree);
   BEGIN NEW(t.name, 1); t.name[0] := 0X;
      t.left := NIL; t.right := NIL
   END Init;

BEGIN Texts.OpenWriter(w)
END Trees.
```

Приложение A: Определение терминов

Целые типы

SHORTINT, INTEGER, LONGINT

Вещественные типы

REAL, LONGREAL

Числовые типы

Целые типы, вещественные типы

Одинаковые типы

Две переменные *a* и *b* с типами *Ta* и *Tb* имеют *одинаковый* тип, если:

1. *Ta* и *Tb* обозначены одним и тем же идентификатором типа, или
2. *Ta* объявлен равным *Tb* в объявлении типа вида *Ta* = *Tb*, или
3. *a* и *b* появляются в одном и том же списке идентификаторов переменных, полей записи или объявлении формальных параметров и не являются открытыми массивами.

Равные типы

Два типа *Ta* и *Tb* равны, если:

1. *Ta* и *Tb* — *одинаковые* типы, или
2. *Ta* и *Tb* — типы открытый массив с *равными* типами элементов, или
3. *Ta* и *Tb* — процедурные типы, чьи списки формальных параметров *совпадают*.

Поглощение типов

Числовые типы *поглощают* (значения) меньших числовых типов согласно следующей иерархии:

LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT

Расширение типов (базовый тип)

В объявлении типа *Tb* = RECORD (*Ta*) ... END, *Tb* — непосредственное расширение *Ta*, а *Ta* — непосредственный базовый тип *Tb*. Тип *Tb* есть расширение типа *Ta* (*Ta* есть базовый тип *Tb*), если:

1. *Ta* и *Tb* — *одинаковые* типы, или
2. *Tb* — непосредственное расширение типа, являющегося расширением *Ta*.

Если *Pa* = POINTER TO *Ta* и *Pb* = POINTER TO *Tb*, то *Pb* есть расширение *Pa* (*Pa* есть базовый тип *Pb*), если *Tb* есть расширение *Ta*.

Совместимость по присваиванию

Выражение *e* типа *Te совместимо по присваиванию* с переменной *v* типа *Tv,* если выполнено одно из следующих условий:

1. *Te* и *Tv* — одинаковые типы;
2. *Te* и *Tv* — числовые типы и *Tv поглощает Te*;
3. *Te* и *Tv* — типы запись, *Te* есть *расширение Tv*, а v имеет динамический тип *Tv*;
4. *Te* и *Tv* — типы указатель и *Te* — расширение *Tv*;
5. *Tv* — тип указатель или процедурный тип, а *e* — NIL;

6. *Tv* — ARRAY n OF CHAR, *e* — строковая константа из *m* символов и *m* < *n*;
7. *Tv* — процедурный тип, а *e* — имя процедуры, чьи формальные параметры *совпадают* с параметрами *Tv*.

ЛАНЬ®

Совместимость массивов

Фактический параметр *a* типа *Ta* является *совместимым массивом* для формального параметра *f* типа *Tf*, если

1. *Tf* и *Ta* — *одинаковые* типы, или
2. *Tf* — открытый массив, *Ta* — любой массив, а их элементы — *совместимые массивы*, или
3. *f* — параметр-значение типа ARRAY OF CHAR, а фактический параметр *a* — строка.

Совместимость выражений

Для данной операции операнды являются *совместимыми выражениями*, если их типы соответствуют следующей таблице (в который указан также тип результата выражения). Символьные массивы, которые сравниваются, должны содержать в качестве ограничителя 0X. Тип T1 должен быть расширением типа T0.

Операция	**Первый операнд**	**Второй операнд**	**Тип результата**
+ – *	*числовой*	*числовой*	наименьший *числовой* тип, поглощающий оба операнда
/	*числовой*	*числовой*	наименьший *вещественный* тип, поглощающий оба операнда
+ – * /	SET	SET	SET
DIV MOD	*целый*	*целый*	наименьший *целый* тип, поглощающий оба операнда
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	*числовой*	*числовой*	BOOLEAN

ЛАНЬ®

	CHAR	CHAR	BOOLEAN
	символьный массив, строка	символьный массив, строка	BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN
	SET	SET	BOOLEAN
	NIL, тип указатель T0 или T1	NIL, тип указатель T0 или T1	BOOLEAN
	процедурный тип T, NIL	процедурный тип T, NIL	BOOLEAN
IN	*целый*	SET	BOOLEAN
IS	тип T0	тип T1	BOOLEAN

Совпадение списков формальных параметров

Два списка формальных параметров *совпадают* если:

1. они имеют одинаковое количество параметров, и
2. они имеют или *одинаковый* тип результата функции или не имеют никакого, и
3. параметры в соответствующих позициях имеют *равные* типы, и
4. параметры в соответствующих позициях — оба или параметры-значения или параметры-переменные.

Приложение B: Синтаксис Оберона-2

Модуль	= MODULE идент ";" [СписокИмпорта] ПослОбъявл [BEGIN ПослОператоров] END идент ".".
СписокИмпорта	= IMPORT [идент ":="] идент {"," [идент ":="] идент} ";".
ПослОбъявл	= { CONST {ОбъявлКонст ";" } | TYPE {ОбъявлТипа ";" } | VAR {ОбъявлПерем ";" }} {ОбъявлПроц ";" | ОпережающееОбъявл ";"}.
ОбъявлКонст	= ИдентОпр "=" КонстВыраж.
ОбъявлТипа	= ИдентОпр "=" Тип.
ОбъявлПерем	= СписокИдент ":" Тип.

<table>
<tr><td>ОбъявлПроц</td><td>= PROCEDURE [Приемник] ИдентОпр [ФормальныеПарам] ";" ПослОбъявл [BEGIN ПослОператоров] END идент.</td></tr>
<tr><td>ОпережающееОбъявл</td><td>= PROCEDURE "^" [Приемник] ИдентОпр [ФормальныеПарам].</td></tr>
<tr><td>ФормальныеПарам</td><td>= "(" [СекцияФП {";" СекцияФП}] ")" [":" УточнИдент].</td></tr>
<tr><td>СекцияФП</td><td>= [VAR] идент {"," идент} ":" Тип.</td></tr>
<tr><td>Приемник</td><td>= "(" [VAR] идент ":" идент ")".</td></tr>
<tr><td>Тип</td><td>= УточнИдент
| ARRAY [КонстВыраж {"," КонстВыраж}] OF Тип
| RECORD ["("УточнИдент")"] СписокПолей {";" СписокПолей} END
| POINTER TO Тип
| PROCEDURE [ФормальныеПарам].</td></tr>
<tr><td>СписокПолей</td><td>= [СписокИдент ":" Тип].</td></tr>
<tr><td>ПослОператоров</td><td>= Оператор {";" Оператор}.</td></tr>
<tr><td>Оператор</td><td>= [Обозначение ":=" Выраж
| Обозначение ["(" [СписокВыраж] ")"]
| IF Выраж THEN ПослОператоров {ELSIF Выраж THEN ПослОператоров} [ELSE ПослОператоров] END
| CASE Выраж OF Вариант {"|" Вариант} [ELSE ПослОператоров] END
| WHILE Выраж DO ПослОператоров END
| REPEAT ПослОператоров UNTIL Выраж
| FOR идент ":=" Выраж TO Выраж [BY КонстВыраж] DO ПослОператоров END
| LOOP ПослОператоров END
| WITH Охрана DO ПослОператоров {"|" Охрана DO ПослОператоров} [ELSE ПослОператоров] END</td></tr>
</table>

<table>
<tr><td></td><td>| EXIT
| RETURN [Выраж]
].</td></tr>
<tr><td>Вариант</td><td>= [МеткиВарианта {"," МеткиВарианта} ":" ПослОператоров].</td></tr>
<tr><td>МеткиВарианта</td><td>= КонстВыраж [".." КонстВыраж].</td></tr>
<tr><td>Охрана</td><td>= УточнИдент ":" УточнИдент.</td></tr>
<tr><td>КонстВыраж</td><td>= Выраж.</td></tr>
<tr><td>Выраж</td><td>= ПростоеВыраж [Отношение ПростоеВыраж].</td></tr>
<tr><td>ПростоеВыраж</td><td>= ["+" | "-"] Слагаемое {ОперСлож Слагаемое}.</td></tr>
<tr><td>Слагаемое</td><td>= Множитель {ОперУмн Множитель}.</td></tr>
<tr><td>Множитель</td><td>= Обозначение ["(" [СписокВыраж] ")"] | число | символ | строка | NIL | Множество | "(" Выраж ")" | "~" Множитель.</td></tr>
<tr><td>Множество</td><td>= "{" [Элемент {"," Элемент}] "}".</td></tr>
<tr><td>Элемент</td><td>= Выраж [".." Выраж].</td></tr>
<tr><td>Отношение</td><td>= "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.</td></tr>
<tr><td>ОперСлож</td><td>= "+" | "-" | OR.</td></tr>
<tr><td>ОперУмн</td><td>= " * " | "/" | DIV | MOD | "&".</td></tr>
<tr><td>Обозначение</td><td>= УточнИдент {"." идент | "[" СписокВыраж "]" | "^"| "(" УточнИдент ")"}.</td></tr>
<tr><td>СписокВыраж</td><td>= Выраж {"," Выраж}.</td></tr>
<tr><td>СписокИдент</td><td>= ИдентОпр {"," ИдентОпр}.</td></tr>
<tr><td>УточнИдент</td><td>= [идент "."] идент.</td></tr>
<tr><td>ИдентОпр</td><td>= идент ["*" | "-"].</td></tr>
</table>

Приложение C: Модуль SYSTEM

Модуль SYSTEM содержит некоторые типы и процедуры, которые необходимы для реализации операций низкого уровня, специфичных для данного компьютера и/или реализации. Они включают, например,

средства для доступа к устройствам, которые управляются компьютером, и средства, позволяющие обойти правила совместимости типов, наложенные определением языка. Настоятельно рекомендуется ограничить использование этих средств специфическими модулями (модулями низкого уровня). Такие модули непременно являются непереносимыми, но легко распознаются по идентификатору SYSTEM, появляющемуся в их списке импорта. Следующие спецификации действительны для реализации Оберон-2 на компьютере Ceres.

Модуль SYSTEM экспортирует тип BYTE со следующими характеристиками: переменным типа BYTE можно присваивать значения переменных типа CHAR или SHORTINT. Если формальный параметр-переменная имеет тип ARRAY OF BYTE, то соответствующий фактический параметр может иметь любой тип.

Другой тип, экспортируемый модулем SYSTEM, — тип PTR. Переменным типа PTR могут быть присвоены значения переменных-указателей любого типа. Если формальный параметр-переменная имеет тип PTR, фактический параметр может быть указателем любого типа.

Процедуры, содержащиеся в модуле SYSTEM, перечислены в таблицах. Большинство их соответствует одиночным командам и компилируются непосредственно в машинный код. О деталях читатель может справиться в описании процессора. В таблице *v* обозначает переменную, *x, y, a* и *n* — выражения, а *T* — тип.

Процедуры-функции

Название	**Типы аргументов**	**Тип результата**	**Функция**
ADR(v)	любой	LONGINT	адрес переменной v
BIT(a, n)	a: LONGINT n: *целый*	BOOLEAN	n-й бит Память[a]
CC(n)	*целая* константа	BOOLEAN	условие n (0 <= n <= 15)
LSH(x, n)	x: *целый*, CHAR, BYTE n: *целый*	совпадает с типом x	логический сдвиг
ROT(x, n)	x: *целый*, CHAR,BYTE n: *целый*	совпадает с типом x	циклический сдвиг
VAL(T, x)	T, x: любого типа	T	x интерпретируется как значение типа T

Собственно процедуры

Название	Типы аргументов	Функция
GET(a, v)	a: LONGINT; v: любой основной тип, указатель, процедурный тип	v := Память[a]
PUT(a, x)	a: LONGINT; x: любой основной тип, указатель, процедурный тип	Память[a] := x
GETREG(n, v)	n: *целая* константа; v: любой основной тип, указатель, процедурный тип	v := Регистр n
PUTREG(n, x)	n: *целая* константа; x: любой основной тип, указатель, процедурный тип	Регистр n := x
MOVE(a0, a1, n)	a0, a1: LONGINT; n: *целый*	Память[a1..a1+n-1] := Память[a0..a0+n-1]
NEW(v, n)	v: любой указатель; n: *целый*	размещает блок памяти размером n байт; присваивает его адрес переменной v

Приложение D: Среда Оберон

Программы на Обероне-2 обычно выполняются в среде, которая обеспечивает *активацию команд*, *сбор мусора*, *динамическую загрузку* модулей и определенные *структуры данных времени выполнения*. Не являясь частью языка, эта среда способствует увеличению мощности Оберона-2 и до некоторой степени подразумевается при определении языка. В приложении D описаны существенные особенности типичной Оберон-среды и даны советы по реализации. Подробности можно найти в [1], [2], и [3].

D1. Команды

Команда — это любая процедура *P*, которая экспортируется модулем *M* и не имеет параметров. Она обозначается *M.P* и может быть активирована под таким именем из оболочки операционной системы. В Обероне, пользователь вызывает команды вместо программ или модулей. Это дает лучшую структуру управления и предостав-

ляет модули с несколькими точками входа. Когда вызывается команда *M.P*, модуль *M* динамически загружается, если он уже не был в памяти (см. прил. D2) и выполняется процедура *P*. Когда *P* завершается, *M* остается загруженным. Все глобальные переменные и структуры данных, которые могут быть достигнуты через глобальные переменные-указатели в *M*, сохраняют значения. Когда *P* (или другая команда *M*) вызывается снова, она может продолжать использовать эти значения.

Следующий модуль демонстрирует использование команд. Он реализует абстрактную структуру данных *Counter,* которая содержит переменную-счетчик и обеспечивает команды для увеличения и печати его значения.

```
MODULE Counter;
  IMPORT Texts, Oberon;

  VAR
    counter: LONGINT;
    w: Texts.Writer;

  PROCEDURE Add*;
  (*получает числовой аргумент из командной строки*)
    VAR s: Texts.Scanner;
  BEGIN
    Texts.OpenScanner
      (s,    Oberon.Par.text, Oberon.Par.pos);
    Texts.Scan(s);
    IF s.class = Texts.Int THEN INC(counter, s.i) END
  END Add;

  PROCEDURE Write*;
  BEGIN
    Texts.WriteInt(w, counter, 5); Texts.WriteLn(w);
    Texts.Append(Oberon.Log, w.buf)
  END Write;

BEGIN counter := 0; Texts.OpenWriter(w)
END Counter.
```

Пользователь может выполнить следующие две команды:

`Counter.Add n`	Добавляет значение n к переменной *counter.*
`Counter.Write`	Выводит текущее значение *counter* на экран.

Так как команды не содержат параметров, они должны получать свои аргументы из операционной системы. Вообще команды вольны брать параметры отовсюду (например из текста после команды, из текущего выбранного фрагмента или из отмеченного окна просмотра). Команда *Add* использует сканер (тип данных, обеспечиваемый Оберон-системой) чтобы читать значение, которое следует за нею в командной строке.

Когда *Counter.Add* вызывается впервые, модуль *Counter* загружается и выполняется его тело. Каждое обращение *Counter.Add n* увеличивает переменную *counter* на *n*. Каждое обращение *Counter.Write* выводит текущее значение *counter* на экран.

Поскольку модуль остается загруженным после выполнения его команд, должен существовать явный способ выгрузить его (например, когда пользователь хочет заменить загруженную версию перекомпилированной версией). Оберон-система содержит команду, позволяющую это сделать.

D2. Динамическая загрузка модулей

Загруженный модуль может вызывать команду незагруженного модуля, задавая ее имя как строку. Специфицированный модуль при этом динамически загружается и выполняется заданная команда. Динамическая загрузка позволяет пользователю запустить программу как небольшой набор базисных модулей и расширять ее, добавляя последующие модули во время выполнения по мере необходимости.

Модуль *M0* может вызвать динамическую загрузку модуля *M1* без того, чтобы импортировать его. *M1* может, конечно, импортировать и использовать *M0*, но *M0* не должен знать о существовании *M1*. *M1* может быть модулем, который спроектирован и реализован намного позже *M0*.

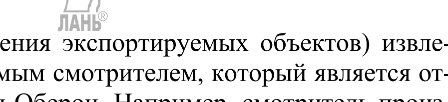
D3. Сбор мусора

В Обероне-2 стандартная процедура NEW используется, чтобы распределить блоки данных в свободной памяти. Нет, однако, никакого способа явно освободить распределенный блок. Взамен Оберон-среда использует сборщик мусора чтобы найти блоки, которые больше не используются и сделать их снова доступными для распределения. Блок считается используемым только если он может быть достигнут через глобальную переменную-указатель по цепочке указате-

лей. Разрыв этой цепочки (например, установкой указателя в NIL) делает блок утилизируемым.

Сборщик мусора освобождает программиста от нетривиальной задачи правильного освобождения структур данных и таким образом помогает избегать ошибок. Возникает, однако, необходимость иметь информацию о динамических данных во время выполнения (см. D5).

D4. Смотритель



Интерфейс модуля (объявления экспортируемых объектов) извлекается из модуля так называемым смотрителем, который является отдельным инструментом среды Оберон. Например, смотритель производит следующий интерфейс модуля *Trees* из п. 11.

```
DEFINITION Trees;
  TYPE
    Tree = POINTER TO Node;
    Node = RECORD
     name: POINTER TO ARRAY OF CHAR;
     PROCEDURE (t: Tree)Insert(name: ARRAY OF CHAR);
     PROCEDURE(t:Tree)Search(name:ARRAY OF CHAR):Tree;
     PROCEDURE (t: Tree) Write;
    END;
  PROCEDURE Init (VAR t: Tree);
END Trees.
```

Для типа запись смотритель также собирает все процедуры, связанные с этим типом, и показывает их заголовки в объявлении типа запись.

D5. Структуры данных времени выполнения

Некоторая информация о записях должна быть доступна во время выполнения. Динамический тип записей необходим для проверки и охраны типа. Таблица с адресами процедур, связанных с записью, необходима для их вызова. Наконец, сборщик мусора нуждается в информации о расположении указателей в динамически распределенных записях. Вся эта информация сохраняется в так называемых дескрипторах типа. Один дескриптор необходим во время выполнения для каждого типа записи. Ниже показана возможная реализация дескрипторов типа.

Динамический тип записи соответствует адресу дескриптора типа. Для динамически распределенных записей этот адрес сохраняется в

так называемом теге типа, который предшествует фактическим данным записи и является невидимым для программиста. Если *t* — переменная типа *CenterTree* (см. пример в п. 6), рисунок D5.1 показывает одну из возможных реализаций структур данных времени выполнения.

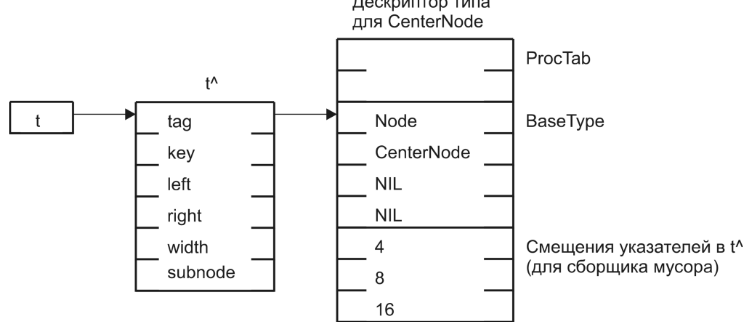


Рис. D5.1. Переменная *t* типа *CenterTree*, запись *t^,* на которую она указывает, и дескриптор типа

Поскольку и таблица адресов процедур и таблица смещений указателей должны иметь фиксированное смещение относительно адреса дескриптора типа, и поскольку они могут расти, когда тип расширяется и добавляются новые процедуры и указатели, то таблицы размещены в противоположных концах дескриптора типа и растут в разных направлениях.

Связанная с типом процедура *t.P* вызывается как *t^.tag^.ProcTab [IndexP].* Индекс таблицы процедур для каждой связанной с типом процедуры известен во время компиляции. Проверка типа *v IS T* транслируется в *v^.tag^.BaseTypes [ExtensionLevelT] = TypeDescrAdrT.* И уровень расширения типа запись (*ExtensionLevelT*), и адрес описателя типа (*TypeDescrAdrT)* известны во время компиляции. Например, уровень расширения *Node* — 0 (этот тип не имеет базового типа), а уровень расширения *CenterNode* — 1.

[1] N. Wirth, J. Gutknecht. The Oberon System // Software Practice and Experience. — 1989. — 19, 9, Sept.

[2] M. Reiser. The Oberon System // User Guide and Programming Manual. — 1991. — Addison-Wesley.

[3] C. Pfister, B. Heeb, J. Templ. Oberon Technical Notes // Report 156, ETH Zürich. — 1991, March.

Литература

1. *Wirth*, *N*. Compiler Construction: This is a slightly revised version of the book published by Addison. — Wesley in 1996. — Zürich, 2005. — 176 с.

2. *Баррон*, *Д*. Введение в языки программирования / пер. с англ. В. А. Серебрякова ; под ред. Ю. М. Баяковского. — М. : Мир, 1980. — 190 с. — (Математическое обеспечение ЭВМ).

3. *Буч*, *Г*. Объектно-ориентированный анализ и проектирование с примерами приложений на Си++. — 2-е изд. — М. : Бином ; СПб. : Невский диалект, 1999. — 560 с. : ил.

4. *Вирт*, *Н*. Алгоритмы + структуры данных = программы : пер. с англ. — М. : Мир, 1985.

5. *Вирт, Н*. Построение компиляторов / пер. с англ. Е. В. Борисов, Л. Н. Чернышов. — М. : ДМК Пресс, 2010. — 192 с. : ил.

6. *Грис*, *Д*. Конструирование компиляторов для цифровых вычислительных машин : пер. с англ. — М. : Мир, 1975. — 544 с.

7. *Гослинг*, *Д*. Язык программирования Java / Д. Гослинг, К. Арнольд. — СПб. : Питер, 1997. — 304 с. : ил.

8. *Зуев*, *Е*. *А*. Язык программирования Си++: этапы эволюции и современное состояние / Е. А. Зуев, А. Н. Кротов, В. А. Сухомлин // тез. докл. Первой российской конференции «Индустрия программирования'96», Москва, 3–4 октября 1996 г. — М., 1997.

9. *Йенсен*, *К*. Паскаль: руководство для пользователя и описание языка / К. Йенсен, Н. Вирт ; пер. с англ., предисл. и послесл. Д. Б. Подшивалова. — М. : Финансы и статистика, 1982. — 151 с. : ил.

10. *Карпов*, *Ю*. *Г*. Теория автоматов : учебник для вузов. — М. ; СПб. ; Н. Новгород [и др.] : Питер , 2002. — 224 с. : ил.

11. *Кауфман*, *В*. *Ш*. Языки программирования. Концепции и принципы. — М. : Радио и связь, 1993. — 432 с. : ил.

12. *Керниган*, *Б.* Язык программирования Си. Задачи по языку Си / Б. Керниган, Д. Ритчи, А. Фьюэр ; пер. с англ. Д. Б. Подшивалова, В. А. Иващенко. — М. : Финансы и статистика, 1985. — 279 с. : ил.

13. Компиляторы: принципы, технологии и инструментарий : пер. с англ. / А. Ахо, Р. Сети, М. Лам, Д. Ульман. — М. : Вильямс, 2008. — 1184 с. : ил.

14. *Матиясевич*, *Ю. В.* 16-разрядная виртуальная ЭВМ, ориентированная на АЯВУ / Ю. В. Матиясевич, А. Н. Терехов // Программирование микропроцессорной техники. — Таллин. — 1984. — С. 68–72.

15. *Матиясевич*, *Ю. В.* Унификация программного обеспечения микроЭВМ на базе виртуальной машин / Ю. В. Матиясевич, А. Н. Терехов, Б. А. Федотов // Автоматика и телемеханика. — 1990. — № 5. — С. 168–175.

16. *Рейуорд-Смит*, *В. Дж.* Теория формальных языков. Вводный курс : пер. с англ. — М. : Радио и связь, 1988. — 128 с. : ил.

17. *Рэдин*, *Дж.* Основные черты NPL — нового языка программирования / Дж. Рэдин, П. Рогуэй ; пер. Т. А. Шаргиной // Современное программирование. Языки для экономических расчетов : сборник статей / пер. с англ. под ред. И. Б. Задыхайло. — М., 1967. — С. 246–276.

18. *Сафонов*, *В. О.* Языки и методы программирования в системе «Эльбрус» / под ред. С. С. Лаврова. — М. : Наука, 1989. — 392 с.

19. *Свердлов С. З.* Введение в методы трансляции : учеб. пособие. — Вологда : Русь, 1994. — 80 с.

20. *Свердлов*, *С. З.* Оберон — воплощение мечты Никлауса Вирта // Компьютерра. — 1996. — № 46 (173). — 25 ноября.

21. *Свердлов*, *С. З.* Маленький большой язык Оберон // PC Week/RE. —1997. — № 35 (109). — 9 сент.

22. *Свердлов*, *С. З.* Арифметика синтаксиса // PC Week / RE. — 1998. — № 42–43.

23. *Свердлов*, *С. З.* Язык программирования Си#: критическая оценка // PC Week/RE. — 2001. — № 20.
24. *Свердлов*, *С. З.* Язык программирования Си#: критическая оценка: (окончание) // PC Week/RE. — 2001. — № 22.
25. *Свердлов*, *С. З.* Языки программирования и методы трансляции : учеб. пособие. — СПб. : Питер, 2007. — 638 с. : ил.
26. *Свердлов*, *С. З.* О структурировании синтаксических диаграмм / С. З. Свердлов, А. А. Хивина // Вестник Вологодского государственного педагогического университета. — 2008. — № 3. — С. 88–92. — (Серия «Физико-математические и естественные науки»).
27. *Свердлов*, *С. З.* Языки и эволюция технологий программмирования : учеб. пособие. — Вологда : ВоГУ, 2016. — 207 с.
28. *Свердлов*, *С. З.* Методы трансляции : учеб. пособие. — Вологда : ВоГУ, 2016. — 234 с.
29. *Страуструп*, *Б.* Язык программирования С++.— М. : Радио и Связь, 1991. — 348 с.
30. *Хантер*, *Р.* Основные концепции компиляторов : пер. с англ. — М. : Вильямс, 2002. — 256 с. : ил.
31. *Хантер*, *Р.* Проектирование и конструирование компиляторов : пер. с англ. / предисл. В. М. Савинкова. — М. : Финансы и статистика, 1984. — 232 с. : ил.
32. *Хопкрофт*, *Д.* Введение в теорию автоматов, языков и вычислений : пер. с англ. / Д. Хопкрофт, Р. Мотвани, Д. Ульман. — 2-е изд. — М. : Вильямс, 2002. — 528 с. : ил.
33. *Эллис*, *М.* Справочное руководство по языку программирования С++ с комментариями / М. Эллис, Б. Страуструп. — М. : Мир, 1992. — 445 с. : ил.

Сергей Залманович СВЕРДЛОВ

ЯЗЫКИ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебное пособие

Издание второе, исправленное

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*
Ответственный редактор *С. В. Макаров*
Корректор *Н. Я. Селиванова*
Выпускающий *Т. А. Быченкова*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com;
196105, Санкт-Петербург, пр. Юрия Гагарина, 1, лит. А.
Тел.: (812) 412-92-72, 336-25-09.
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 19.03.19.
Бумага офсетная. Гарнитура Школьная. Формат 84×108 $^{1}/_{32}$.
Печать офсетная. Усл. п. л. 29,61. Тираж 100 экз.

Заказ № 212-19.

Отпечатано в полном соответствии
с качеством предоставленного оригинал-макета
в АО «Т8 Издательские технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.