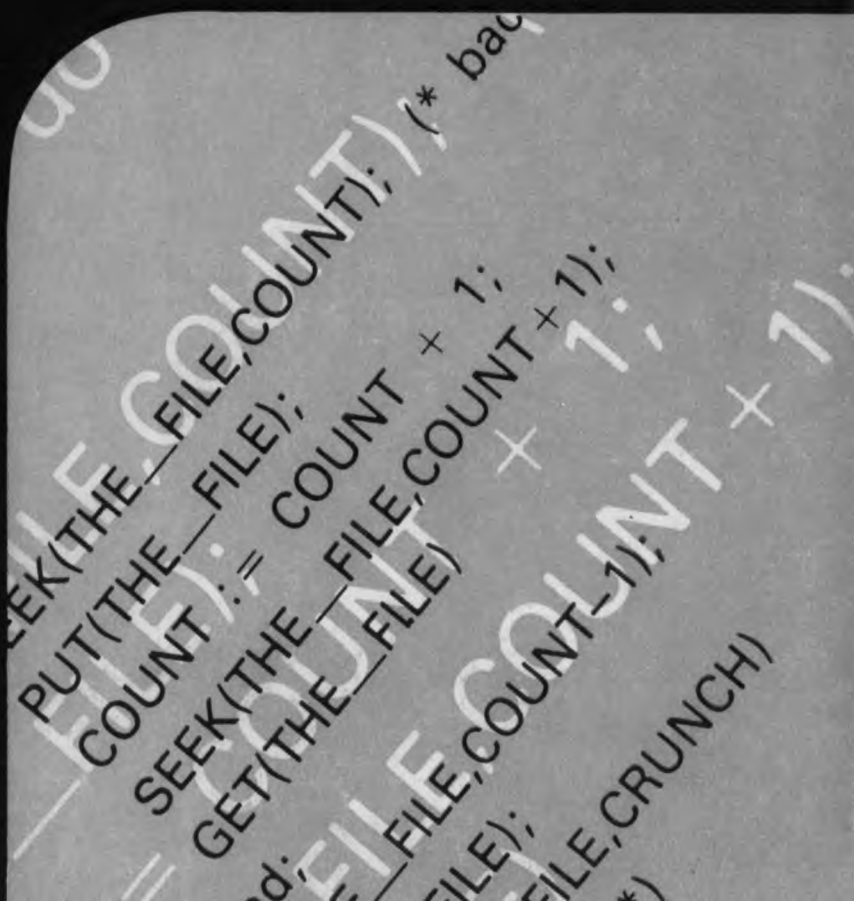
# Д. Прайс
# Программирование на языке Паскаль
## Практическое руководство



# Программирование на языке Паскаль

D. Price

# UCSD Pascal A Considerate Approach

Prentice-Hall, Inc.,
Englewood Cliffs, 1983

# Д. Прайс

# Программирование на языке Паскаль: Практическое руководство

Перевод с английского
**А. П. Пшокина**

под редакцией

канд. техн. наук **О. Н. Перминова**


Москва «Мир» 1987

ББК 32.973-01
П 68
УДК 681.3

**Прайс Д.**

П 68 Программирование на языке Паскаль: Практическое руководство. Пер. с англ. — М.: Мир, 1987. — 232 с., ил.

В книге крупного специалиста из США излагается диалект языка Паскаль, разработанный в Калифорнийском университете г. Сан-Диего под руководством К. Боулса. Детально описываются все конструкции языка, приводятся многочисленные примеры программ. Особое внимание уделяется стилю программирования и вопросам проектирования программ. Книга имеет практическую направленность, что делает ее общедоступной. Многие примеры представляют несомненную практическую ценность, и их можно использовать как фрагменты проектируемых пользователем программ.

Для широкого круга пользователей, не имеющих специальной подготовки в области программирования, а также студентов, аспирантов и специалистов в области программного обеспечения ЭВМ.

П $\frac{2405000000-131}{041(01)-87}$ 172—87, ч. 1 ББК 32.973-01

*Редакция литературы по информатике и робототехнике*

## Предисловие редактора перевода

Предлагаемая читателю книга является одной из немногих в СССР публикаций, посвященных языку программирования Паскаль, и первой, полностью посвященной одному из диалектов этого языка. В оригинале он называется UCSD-Pascal (язык Паскаль Калифорнийского университета в г. Сан-Диего). Мы будем называть его УКСД-Паскаль.

В языке УКСД-Паскаль учтены многие критические замечания в адрес исходного языка, разработанного Н. Виртом. Основная цель данного расширения — сделать язык более удобным для практического использования. Программисты-практики сразу же оценят новые возможности. Многие действия, которые при программировании на стандартном языке Паскаль приходилось каким-то образом моделировать, в рамках данного расширения выражаются легко и просто.

Интересно выделить наиболее принципиальные отличия языка УКСД-Паскаль от стандарта языка. При этом за точку отсчета будем брать документ Британского института стандартов BS 6192 : 1982, являющийся также и международным стандартом. Основные черты расширения следующие:

1. В идентификаторах разрешается использовать символ подчеркивания, что особенно полезно при записи имен, состоящих из нескольких слов.

2. Введен тип данных STRING и семь операций над данными этого типа: CONCAT (сцепление строк), POS (проверка вхождения одной строки в другую), COPY (выделение подстроки заданной длины), DELETE (удаление заданной подстроки), INSERT (вставка одной строки в другую), STR (преобразование аргумента целого типа в строку), LENGTH (подсчет числа литер в строке).

3. Введен оператор EXIT, позволяющий завершить выполнение процедуры, функции или программы.

4. Для текстовых файлов введены два стандартных

типа TEXT и INTERACTIVE. Соответственно изменены и процедуры RESET и REWRITE. Введены средства контроля выполнения этих процедур.

5. Закрытие текстовых файлов осуществляется с помощью процедуры CLOSE и при этом возможны различные режимы.

6. Для работы с файлами помимо последовательного доступа вводится также режим произвольного доступа, с помощью которого можно обращаться к компонентам файла с указанным номером.

7. Имеется возможность работать с целыми числами длиной до 36 цифр. Над ними можно выполнять все операции целочисленной арифметики, за исключением нахождения остатка от целочисленного деления.

Простое перечисление дополнительных средств языка УКСД-Паскаль позволяет сделать вывод, что область применения его существенно расширяется по сравнению со стандартным языком. При этом обращает на себя внимание тщательная проработка введенных языковых средств. И вместе с тем язык УКСД-Паскаль в отличие от других версий, например Паскаль-плюс, все-таки остается языком Паскаль.

Несколько слов о книге в целом. Она написана очень легко и свободно. При чтении не возникает почти никаких затруднений в восприятии материала. Все конструкции языка выделены в тексте, поэтому книгу можно использовать и как справочное пособие. Примеры детально объяснены, и в них используются многие конструкции, которые можно копировать в свои программы. Однако в монографии нет уже ставших традиционными синтаксических диаграмм, описывающих правила языка, и строгих семантических правил, регламентирующих использование тех или иных конструкций. Для читателей, которых интересуют мельчайшие подробности языка, можно рекомендовать другие источники, например: Перминов О. Н. Язык программирования Паскаль. — М.: Радио и связь, 1983.

Большое внимание уделено стилю написания программ. Вопрос действительно очень важный. Начинающему программисту эти страницы будут особенно полезны. Тщательная их проработка поможет избежать многих неприятных моментов, через которые методом проб и ошибок прошло уже не одно поколение программистов. Опытному программисту не все может показаться вер-

ным. Вместе с тем альтернативная точка зрения дает возможность еще раз задуматься над собственным стилем программирования и, быть может, внести в него некоторые коррективы.

В заключение о переводе. Прошло уже пять лет с момента выхода первой книги о языке Паскаль на русском языке (Грогоно П. Программирование на языке Паскаль. — М.: Мир, 1982). Многие программисты используют этот язык в практической деятельности. Уже сложился не только терминологический стиль, но даже и жаргон языка Паскаль. Мы старались не отклоняться от традиционной терминологии, однако некоторые отличия все-таки имеются.

Особо следует отметить следующее. Данная книга полностью посвящена языку УКСД-Паскаль. Поэтому во избежание постоянного повторения названия УКСД-Паскаль мы использовали более короткое и привычное — просто Паскаль. В рамках данной книги это не вызывает недоразумений и двусмысленностей. В более широком же контексте следует различать основной язык и его диалекты.

Хочется надеяться, что данная книга окажется полезной как начинающим программистам, так и тем, кто уже знаком с языком Паскаль.

*О. Н. Перминов*

# Глава 1
# Введение

Языки ЭВМ выступают как средство общения. Если человеческий язык — это средство общения людей, то язык ЭВМ — средство общения человека и машины. Язык Паскаль был разработан профессором Никлаусом Виртом (Швейцарская Высшая техническая школа, г. Цюрих). Несмотря на то что он был разработан как средство для обучения хорошей технике программирования, впоследствии он стал популярным и вне сферы образования.

По мере увеличения популярности языка Паскаль стали возникать его «диалекты». Некоторые из них близки к оригиналу, в то время как другие отличаются от него весьма существенно. Излагаемый в данной книге язык Паскаль, разработанный в Калифорнийском университете (г. Сан-Диего), попадает во вторую категорию. Под руководством профессора Кяннета Боулса разработчики языка расширили его возможности средствами, которые полезны во многих областях применения.

В отличие от человеческого язык ЭВМ не допускает двусмысленностей и неопределенностей. Каждый язык ЭВМ имеет строго определенную грамматику, называемую *синтаксисом.* Если предложение программы (оператор) не соответствует синтаксису языка, то оно не имеет смысла. С другой стороны, синтаксически правильный оператор имеет однозначную трактовку.

Программа для ЭВМ — это просто последовательность операторов. Операторы представляют собой команды ЭВМ, следуя которым она решает задачу. Метод, используя который программа решает задачу, называется *алгоритмом*; дело программиста выбрать алгоритм, чтобы составить по нему программу. Алгоритм может представлять собой просто математическое уравнение (например, квадратное уравнение) или сложную процедуру. В любом случае представление алгоритма в виде программы является задачей программиста.

Нет ничего удивительного в том, что различные программисты придерживаются различных точек зрения

при написании программ. Некоторые стараются добиться, чтобы программа содержала минимальное число операторов или выполнялась за наименьшее время. Точка зрения на программирование, представленная в этой книге, основывается на другом критерии качества программы. Хорошая программа в соответствии с этим критерием — это программа, написанная так, что она понятна другим людям. Более подробно это означает, что программист должен писать программу так, чтобы:

1) ее было легко читать и
2) ее было легко использовать.

В этой книге хорошим считается программирование, удовлетворяющее указанным критериям. Если вы «воспитаны» на представлениях, что программирование — дисциплина, ориентированная на машину, предлагаемый здесь взгляд может показаться незаслуживающим внимания. Однако программирование для ЭВМ — не настолько машиноориентированная процедура, как это могло бы показаться. Поскольку ЭВМ существуют для обслуживания людей, программирование для ЭВМ принципиально «человекоориентированная» дисциплина по определению. Важной установкой программиста, следовательно, должно быть стремление удовлетворить потребность людей, которые будут читать и использовать его программу.

Цели, отмеченные выше, могут показаться очевидными, однако в реальной жизни они не всегда легко достижимы. Основная задача, состоящая в написании хорошей программы, требует для своего решения серьезного, продуманного отношения. При разработке программы часто возникает искушение принести ясность в жертву незначительному повышению эффективности. К счастью, язык Паскаль обладает хорошими средствами для проектирования понятных программ. Его систематизированная структура налагает серьезные ограничения на дисциплину программирования[1] и в то же время позволяет расширять язык за счет введения новых возможностей, которые удовлетворяют специальным применениям. Короче говоря, язык Паскаль облегчает программисту задачу написания хороших программ.

---

[1] Тем не менее следует вспомнить «аксиому» программирования: «Нет, не было и, скорее всего, не будет языка программирования, который, хотя бы в малейшей степени, гарантировал нас от написания плохих программ». Наилучшие средства в неумелых руках могут дать эффект, прямо противоположный ожидаемому. — *Прим. ред.*

# Глава 2

# Простая программа

## 2.1. Как выглядит программа?

Предположим, вы хотите написать программу, которая вычисляет сумму двух чисел. Как сделать это? Как будет выглядеть написанная программа? Чтобы вы получили представление о том, как программируют на языке Паскаль, приведем пример 2.1.

---

**Пример 2.1. Сумма двух чисел**

```
program SUMS;
var
        ADDEND1, ADDEND2, RESULT : INTEGER;
begin
        ADDEND1 : = 3;
        ADDEND2 : = 4;
        RESULT : = ADDEND1 + ADDEND2;
        WRITELN (RESULT)
end.
```

---

Что означает эта запись? Для ответа на этот вопрос разобьем программу на более мелкие части. Первая строка называется *заголовком программы.* В заголовке указывают имя программы. В данном случае программа названа SUMS. Если мы выбрали имя ADDER, то запись выглядела бы следующим образом:

```
program ADDER;
```

В примере 2.1, как и в последующих примерах, некоторые слова напечатаны строчными буквами. Эти слова называются *зарезервированными словами,* так как в каждое из этих слов вкладывают особый смысл. Например, слово *program* говорит о том, что читается строка заголовка. Другие зарезервированные слова также имеют определенный смысл. (Когда текст программы предназначен для обработки на вычислительной машине,

не возникает необходимости производить какие-либо типографские выделения. Прописные буквы для представления зарезервированных слов служат для обеспечения большей наглядности представленных примеров программ.)

## 2.2. Описание переменных

Следующие после заголовка две строки в примере программы 2.1 представляют *раздел описания переменных.* Он указывает вычислительной машине, сколько переменных используется в программе, какие имена у этих переменных и данные какого типа будут храниться в этих переменных. В программе, названной SUMS, описываются три переменные с именами ADDEND1, ADDEND2 и RESULT. Все переменные имеют тип INTEGER, который означает, что переменные могут принимать только целочисленные значения.

Какой смысл вкладывается в понятие «переменная?» Будем рассматривать переменную как «черный ящик», в котором могут храниться данные. Например, переменная типа INTEGER может содержать только целые числа. Другие типы данных будут рассмотрены в последующих главах, а до тех пор во всех примерах программ используются только переменные целого типа.

На имена, присваемые переменным, накладываются некоторые ограничения. Имена могут нести смысловую нагрузку, например RESULT, но могут и не нести, например ZBLICN. Как видно, использование осмысленных имен предпочтительнее, так как делает программу более простой для понимания. Практически всегда стараются выбирать имена переменных такими, чтобы они отражали назначение переменной. Если следовать данному правилу, то никогда не возникнет вопрос, какая переменная соответствует решаемой задаче. Приведем наиболее важные ограничения, имеющие место при выборе имен переменных.

1. Имена переменных не могут быть зарезервированными словами. Таким образом, не может быть объявлена переменная, вызванная PROGRAM или VAR (в приложении А приведен полный список зарезервированных слов).

2. Имена переменных должны начинаться с буквы.

Следовательно, имя ADDEND1 является допустимым, а имя 1ADDEND не является таковым.

3. Имена переменных могут быть любой длины, но значительными являются только первые восемь литер[1]. Отсюда следует, что если в двух именах совпадают первые восемь литер, то они считаются идентичными. Следовательно, имена ACCOUNTSPAYABLE и ACCOUNTSRECEIVABLE в действительности представляют одну и ту же переменную.

Для лучшего восприятия имен можно использовать символ подчеркивания в именах переменных. Подчеркивания особенно полезны, когда имена состоят из нескольких слов, например ACCOUNTS_PAYABLE или ELECTRON_MASS. Символ подчеркивания в имени переменной не воспринимается ЭВМ, и с позиции вычислительной машины имена TIME_OF_DAY и TIMEOFDAY идентичны.

После того как описаны все переменные, которые будут использоваться в программе, с ними можно работать — присваивать им значение, использовать их в вычислениях и распечатывать их значения. Таким образом, мы подошли к последнему разделу программы примера — *телу программы*.

## 2.3. Тело программы

Это та часть программы, где описываются выполняемые действия. Разделы программы, рассмотренные ранее, — заголовок и раздел описания переменных — были подготовкой вычислительной машины (и читателя) к восприятию *тела программы*. В теле программы указывается последовательность действий, которые должны быть выполнены ЭВМ.

Тело программы заключается между двумя зарезервированными словами: *begin* и *end*. По своему функциональному назначению эти слова аналогичны открывающей и закрывающей скобкам соответственно. Если необходимо отметить конец программы, то после *end* становится точка. Из дальнейшего изложения станет понятна необходимость следования этому правилу, так как конструкция *begin...end* используется и для других целей. Чтобы не затруднять понимание, временно не

[1] В стандарте языка Паскаль этого ограничения нет. — *Прим. ред.*

используется один из важных аспектов пунктуации языка Паскаль, связанный с применением литеры «точка с запятой».

Первые три строчки тела примера программы являются *операторами присваивания.* Цель оператора присваивания заключается в задании некоторых значений переменным. Этими значениями могут быть математические выражения, например ADDEND1 + ADDEND2. В целочисленной арифметике используется и ряд других операций:

| | |
|---|---|
| − | вычитание |
| * | умножение |
| div | целочисленное деление |
| mod | остаток от деления |
| ABS | абсолютная величина |
| SQR | квадрат числа |

На основе этих операций строятся выражения требуемого уровня сложности. Ниже приведены примеры целочисленных выражений на языке Паскаль и значения этих выражений:

| | |
|---|---|
| 9−4 | (равно 5) |
| 6 * 7 | (равно 42) |
| 20 − 4 + 1 | (равно 17) |
| (6 + 3) * 5 | (равно 45) |
| 2 * (−2) | (равно −4) |
| 9 div 3 | (равно 3) |
| 10 div 3 | (равно 3) |
| 9 mod 3 | (равно 0) |
| 10 mod 3 | (равно 1) |
| 11 mod 3 | (равно 2) |
| ABS (−5) | (равно 5) |
| ABS (3 * 4) | (равно 12) |
| SQR (4) | (равно 16) |
| SQR (6) − 1 | (равно 35) |

В выражениях, состоящих из ряда операций, последовательность выполнения операций определяется стандартными алгебраическими правилами следования. Операции умножения и деления выполняются ранее операций сложения и вычитания, так как умножение и деление имеют приоритет перед операциями сложения и вычитания. Как и в любых математических записях, можно изменить порядок выполнения операций, применив скобки.

Таким образом, (1 + 2) * 3 = 9. *но* 1 + 2 * 3 = 7.

Если знак плюс или минус встречается после другого знака операции, то их следует разделить с помощью скобок.

Ниже приведен пример неправильной записи выражения:

```
5 * - SCORE
```

Для того чтобы исправить это выражение, его следует переписать следующим образом:

```
5 * (-SCORE)
```

Среди целочисленных операций следует пояснить смысл операции *mod*. Операция *mod* (сокращение от *modulo*) дает возможность найти остаток от выполнения операции целочисленного деления. Для двух положительных целых чисел *i* и *j* выражение *i mod j* эквивалентно выражению

```
i - (i div j)* j
```

Операция *mod* является очень удобной. Операция *div* вычисляет только целую часть частного, а его остаток можно найти с помощью операции *mod*. У операции *mod* имеется и ряд других применений. Например, если x mod 2 равен 0, то x четно, если x mod 2 равен 1, то нечетно.

Как выполняется присваивание? Делается это просто: переменной слева от знака : = присваивается значение выражения, стоящего справа от знака : = . В примере программы переменной ADDEND1 присваивается значение 3, а ADDEND2 — значение 4. Переменная RESULT принимает значение, равное 7, так как (ADDEND1 + + ADDEND2) = (3 + 4) = 7.

Переменная рассматривается как *неопределенная,* если она описана, но значение переменной еще не задано. Неопределенная переменная не имеет значения. При попытке использовать в вычислениях неопределенную переменную работа программы будет прервана. Пример 2,2, представленный ниже, демонстрирует неотлаженную программу, в которой делается попытка использовать переменную ADDEND2 до того, как определяется ее значение.

**Пример 2.2. Программа, которая не будет работать**

```
program NOGOOD;
var
      ADDEND1, ADDEND2, RESULT : INTEGER;
begin
      ADDEND1 : = 3;
      RESULT : = ADDEND1 + ADDEND2;          *
      ADDEND2 : = 4;
      WRITELN (RESULT)
end.
```

В этой программе будет вырабатываться условие ошибки при выполнении оператора, помеченного *, так как переменной ADDEND2 еще не присвоено значение.

Некоторые операторы присваивания выглядят странно, несмотря на то, что с позиции вычислительной машины такие операторы вполне допустимы[1]. Прикрыв рукой следующий абзац, постарайтесь определить действие этого оператора присваивания:

AGE : = AGE + 1

Как вы и предполагали, этот оператор увеличивает значение переменной AGE на 1. Предположим, AGE равно 5. Встретив этот оператор присваивания, ЭВМ вычислит выражение (AGE + 1) = (5 + 1) = 6. Таким образом, новое значение AGE равно 6. Аналогично AGE: = AGE − 1 будет уменьшать на 1 значение AGE. (Конечно, если AGE предварительно не было присвоено значение, то эти операторы приведут к состоянию ошибки «не определено значение переменной», которое рассмотрено выше.)

Что происходит со старым значением переменной, когда ей присваивается новое значение? Оно просто затирается. Поскольку переменная типа INTEGER может хранить только одно число, то выполнение оператора присваивания приводит к потере предыдущего значения переменной. Переменная всегда содержит результат выполнения последнего оператора присваивания.

[1] Например, для математика, не знакомого с программированием, нижеследующий оператор присваивания может показаться уравнением, не имеющим решения. — *Прим. ред.*

В примере 2.1 остается нерассмотренным только один оператор. Несмотря на то что WRITELN не относится к зарезервированным словам, этот оператор[1] языка Паскаль имеет специальное значение. Оператор WRITELN предписывает вычислительной машине отобразить определенные данные на терминале ввода-вывода пользователя. В качестве терминала может использоваться пишущая машинка или дисплей. Список элементов, которые следует вывести на устройство отображения информации, приводится внутри круглых скобок. Элементом списка могут быть число, переменная, математическое выражение или обычный текст. Примеры правильного написания операторов приведены ниже:

```
WRITELN (52,52 * 2,52 * 3)
WRITELN (RESULT * 2)
WRITELN (ABS (DEFICIT))
WRITELN (DIVISOR div DIVIDEND)
WRITELN ('Это предложение будет напечатано дословно.')
```

Предполагая, что переменные RESULT, DEFICIT, DIVISOR и DIVIDEND описаны и им присвоены значения, представленные операторы WRITELN распечатают результаты приведенных выражений. В последнем примере показано использование символьных строк в операторе WRITELN. Все символы между одиночными апострофами будут напечатаны в том виде, как они представлены в тексте. В рамках одного оператора WRITELN можно наряду с символьными строками использовать выражения, например:

```
WRITELN ('Площадь данного прямоугольника равна',
LENGTH * WIDTH)
```

Более подробно оператор WRITELN и другие операторы ввода-вывода будут обсуждаться в третьей главе. Рассмотрим применение литеры «точка с запятой».

---

[1] В языке Паскаль нет специальных операторов ввода-вывода. Для этих целей используются обращения к стандартным процедурам READ, WRITE, WRITELN, READLN. Однако по аналогии с другими языками такие обращения часто называют *операторами.* Так будем называть их и мы. — *Прим. ред.*

## 2.4. Для чего служит точка с запятой?

Точка с запятой используется в языке Паскаль для разделения операторов. В общем случае каждый оператор должен оканчиваться точкой с запятой. Имеется только несколько исключений из этого правила. Одно из наиболее часто используемых таких исключений, которое уже упоминалось, — это конструкция *begin ... end.*

За зарезервированным словом *begin* никогда не следует точка с запятой, так же как она никогда не ставится перед зарезервированным словом *end.* Для того чтобы понять, почему конструкция *begin ... end* является специальным случаем, рассмотрим структуру блока *begin ... end*

*begin* $S_1$; $S_2$; $S_3$; ... $S_{n-1}$; $S_n$ *end*

Вспомните, что *begin* и *end* аналогичны скобкам. В приведенном примере каждый символ *S* представляет единственный оператор. Вы не будете ставить дополнительных точек с запятой в записи

($S_1$; $S_2$; $S_3$; $S_4$)

По тем же соображениям не следует ставить точку с запятой после *begin* и перед *end.* Отметим, что эти правила соблюдаются в примере 2.1. Чтобы лучше познакомиться с использованием точки с запятой, будьте внимательны при чтении примеров программ.

## 2.5. Синтаксические ошибки

Подобно человеческому языку, который обладает своей формальной грамматикой, машинные языки также имеют обязательные грамматические правила, или *синтаксис* языка. Каждый машинный язык имеет собственный синтаксис. Излишне напоминать о том, что программист, использующий язык Паскаль (или какой-то другой машинный язык), должен строго следовать синтаксическим правилам языка. Наиболее распространенные синтаксические ошибки связаны с ошибками в применении точки с запятой, орфографическими ошибками при написании зарезервированных слов[1] и с другими

[1] Некоторые компиляторы исправляют ошибки в зарезервированных словах, понятные из контекста. — *Прим. ред.*

на первый взгляд малозначительными нарушениями. Начинающий программист обычно спешит спросить: «Почему вычислительная машина не может исправить мои синтаксические ошибки?»

Более опытный программист ответит: «Но вычислительной машине, прежде чем исправить ваши ошибки, необходимо знать, что вы хотели сказать». Другими словами, ЭВМ еще не могут читать мысли. Диагностика и исправление ошибок — не механический процесс. Как будет видно из дальнейшего, этот процесс часто является самой кропотливой фазой разработки программы. Существует даже специальный термин для обозначения этой фазы. Процесс диагностики и исправления ошибок называется *отладкой.* Во время отладки исправляются не только синтаксические ошибки, но также ошибки, которые проявляются на этапе выполнения программы.

---

**Оператор присваивания**

Знак:

:=

Общая форма записи:

*переменная:= выражение*

Оператор присваивания устанавливает значение *переменной* равным величине, полученной в результате вычисления *выражения.* Оба элемента, *переменная* и *выражение,* должны быть одного типа. Например, если переменная имеет тип INTEGER, то выражение также должно формировать результат целого типа.

Примеры:

```
COUNT := COUNT + 1

PAY : = WACE * HOURS
FORCE : = MASS * ACCEL

RETAILPRICE : = WHOLESALEPRICE * 2
```

---

В этой книге основные сведения о синтаксисе языка Паскаль будут выделены в тексте линейками аналогично приведенному выше описанию. Все определения выделены из текста и поэтому при необходимости могут быть легко найдены и использованы.

ВОПРОСЫ[1]

1. Напишите программу, которая вычисляет площадь треугольника, основание которого равно 3 см и высота 5 см. Следует обеспечить достаточную информативность выводимых результатов работы программы. Имена переменных должны отражать смысл переменных.

2. Предположим, что в языке Паскаль имена переменных могут быть односимвольные. Таким образом, переменные R и V допустимы, а имена RATE и VELOCITY нет. Какие нежелательные последствия могли бы иметь эти ограничения?

3. Используя выражения целого типа, напишите операторы присваивания, которые эквивалентны следующим формулам:

$A = LW$

$M = R^3 / P^2$

$$S = \frac{Y_1 - Y_2}{X_1 - X_2}$$

$X = |A|$

4. Исправьте в приведенной ниже программе ошибки:

```
program BUGLADEN;
var
      BEGIN : INTEGER;
begin;
      TEMP : = 13;
      BEGIN : = SQR (TEMP);
      WRITELN ('Тринадцать в квадрате равно', BEGIN
end.
```

[1] Здесь и далее ответы на вопросы, помеченные звездочкой, помещены в приложении.

# Глава 3
# Ввод и вывод информации

## 3.1. Операторы WRITE и WRITELN

Использование оператора WRITELN уже рассматривалось. В данном разделе остановимся на более подробном изучении правил применения операторов WRITE и WRITELN. Ниже дано краткое описание этих операторов.

---

WRITE и WRITELN
Общая форма записи:

WRITE *(список выражений)*
WRITELN *(список выражений)*

Эти операторы используются для вывода информации на терминал ввода-вывода пользователя.
Примеры:

```
WRITE ('Этот текст является примером символьной строки.')
WRITELN ('Ваш налоговый счет за год составляет $', TAX : 5)
WRITELN (ACRES : 4, 'акров не обработаны.')
```

---

Действия операторов WRITE и WRITELN во многом идентичны, за одним исключением. Рассмотрим действие двух операторов.

```
WRITELN ('Первая строка'.)
WRITELN ('Вторая строка'.)
```

Эта последовательность операторов приведет к появлению на выходе следующего текста:

```
Первая строка.
Вторая строка.
```

Выполнение первого оператора WRITELN приводит к тому, что последующий вывод данных будет начинаться с новой строки. Это достигается за счет того,

что оператор WRITELN после распечатки списка выражений автоматически переводит каретку или курсор в начало следующей строки. Если в приведенном выше примере оба оператора WRITELN заменить на WRITE, то будет напечатано:

Первая строка. Вторая строка.

Данные, выводимые обоими операторами, печатаются на одной строке. Оператор WRITE после своего выполнения не осуществляет позиционирования печатающего механизма на начало строки. За исключением нескольких случаев, в дальнейшем будет использоваться оператор WRITELN. (Оператор WRITE можно, например, использовать, когда нужно поместить в одну строку список из нескольких выражений.)

В некоторых случаях возникает необходимость напечатать апостроф в середине символьной строки. Чтобы сделать это, апостроф записывают дважды. При печати строки два последовательных апострофа представляются одним. Например, для печати строки

*What's your name?*

Мы должны записать

WRITELN (*'What''s your name?'*)

При распечатке целочисленных значений можно задать минимальный размер поля для каждой величины. Размер поля выражается числом позиций, отводимых для печати этого значения. Выполняя печать некоторого значения, вычислительная машина определяет требуемое число позиций. (Например, числу 999 будут требоваться три позиции.) Если указанный минимальный размер поля больше необходимого, то слева от значения вставляются пробелы. В случаях когда минимальный размер поля меньше требуемого или минимальный размер не указан, значение печатается без каких-либо дополнительных пробелов. Приведем оператор WRITELN, в котором определен размер поля:

WRITELN ('Дефицит составляет', DEBTS : 12).

Если DEBTS имеет значение 1500, то результат выполнения оператора будет выглядеть следующим образом:

Дефицит составляет ␣␣␣␣␣␣␣␣ 1500.

В операторе WRITELN число за двоеточием указывает

желаемый минимум размера поля. Для обеспечения размера поля в двенадцать позиций в приведенном примере производится дополнение значения восемью пробелами. В программе из примера 3.1 определены минимальные размеры полей всех выводимых целочисленных значений.

---

**Пример 3.1. Среднее значение трех чисел**

```
program AVERAGE;
var
      FIRST, SECOND, THIRD : INTEGER;
      SUM : INTEGER;
begin
      FIRST : = 5;
      SECOND : = 17;
      THIRD : = 8;
      SUM : = FIRST + SECOND + THIRD;
      WRITELN ('Среднее значение', FIRST : 4, ', ',
                                   SECOND : 4);
   WRITELN (' и ', THIRD : 4,' равно ', (SUM div 3) : 3)
end.
```

---

Имеется возможность распечатать строку одних пробелов. Это может потребоваться для разделения данных. В языке Паскаль это достигается выполнением простого оператора:

```
WRITELN
```

## 3.2. READ и READLN

Рассмотрим программу, приведенную в примере 3.1. Цель данной программы заключается в нахождении среднего значения трех чисел. В результате выполнения программы вычисляется среднее значение трех чисел: 5, 17 и 8. Однако для того, чтобы использовать программу для нахождения среднего значения других чисел, придется переписать несколько операторов присваивания.

Кому захочется всякий раз выполнять эту работу? Основное назначение ЭВМ — сэкономить человеческий труд, а не создавать дополнительные трудности. Поэтому необходимо обеспечить возможность, однажды написав программу, многократно использовать ее в различных

ситуациях. Относительно программы AVERAGE это выражается в возможности определять желаемые числа, не изменяя операторов самой программы. Такая гибкость обеспечивается в языке Паскаль операторами READ и READLN.

---

**READ и READLN**

Общая форма записи;

READ *(переменная, переменная...)*

READLN *(переменная, переменная...)*

Эти операторы обеспечивают ввод информации пользователя с терминала ввода-вывода.

Примеры:

```
READ (HOURS)
READ (DAY, MONTH, YEAR)
READLN (RADIUS)
READLN (GROSSPAY, NETPAY)
```

---

Теперь можно усовершенствовать программу AVERAGE так, что она будет принимать новые значения переменных при каждом выполнении. Текст программы AVERAGE после внесения изменений показан в примере 3.2. Значения переменных FIRST, SECOND и THIRD в новом варианте программы вводятся с терминала пользователя, не используя для этой цели операторы присваивания.

---

**Пример 3.2. Интерактивная программа вычисления среднего значения**

```
program AVERAGE2;
var
      FIRST, SECOND, THIRD : INTEGER;
      SUM : INTEGER;
begin
      WRITELN ('Введите три числа, среднее значение ко
      торых вы хотите получить:');
      READ (FIRST, SECOND, THIRD);
      SUM : = FIRST + SECOND + THIRD;
      WRITELN ('Среднее значение ', FIRST : 4, ', ',
                                    SECOND : 4, ' и ');
      WRITELN (THIRD : 4, ' равно ', (SUM div 3) :3)
end.
```

---

Новая программа AVERAGE2 лучше предыдущей. Приступив к выполнению оператора READ, вычислительная машина переходит в состояние ожидания ввода информации. После ввода трех чисел каждой из переменных, приведенных в списке оператора READ, будет присвоено соответствующее значение. Переменная FIRST примет значение первого введенного числа и т. д. Ниже продемонстрируем работу программы AVERAGE2:

```
Введите три числа, среднее значение которых вы хотите получить:  3  52  -27
Среднее значение  3,  52  и  -27  равно  9
```

Чтобы отделить три вводимых числа друг от друга, в данном случае были использованы дополнительные пробелы. Несмотря на то что для этой цели обычно используются пробелы или запятые, разрешено применять любые литеры, за исключением знаков плюс или минус. Окончание ввода вычислительная машина распознает по литере «конец строки», которой заканчивают вводимую строку. В зависимости от типа используемого терминала эта клавиша может быть помечена «возврат каретки» или «ввод».

Хотя в работе операторов READ и READLN существуют некоторые отличия, до определенной поры они не являются существенными. При чтении и записи числовых значений с терминала ввода-вывода в использовании операторов READ и READLN нет различий. (Особенности использования READ и READLN будут рассмотрены позже.)

### 3.3. Проектирование вывода

Качество программы во многом определяется не только содержанием выводимой информации, но и ее формой. Одно дело написать работоспособную программу и совсем другое — написать программу с хорошо спроектированным выводом. Хорошие результаты получаются, если руководствоваться правилом: тот, кто использует программу, не ясновидец.

Безусловно, это разумное предположение. Человек, который использует чужую программу, не должен детально разбираться в том, что́ происходит в ней. Следовательно, неразумно ожидать от него, что он правильно поймет бессвязный вывод с непонятными сокращениями или неоднозначностями.

**Пример 3.3. Программа с неудачным вариантом вывода**

```
program AVOID;
var
        FIRST, SECOND, THIRD : INTEGER;
        SUM : INTEGER;
begin
        WRITELN ('Задайте числа');
        READ (FIRST, SECOND, THIRD);
        SUM : = FIRST + SECOND + THIRD;
        WRITELN (SUM div 3)
end.
```

С точки зрения пользователя, в программе примера 3.3 не все ясно. Сравним текст предложения ('Задайте числа') с аналогичной фразой из примера 3.2. При прочтении новой редакции фразы неосведомленный пользователь, вероятно, будет теряться в догадках: «Сколько чисел я должен ввести?», «Для чего эти числа будут использованы?»

Представление результата вычислений тоже страдает недостатками. По завершению вычислений программа просто печатает число, не сопровождая его какими-либо пояснениями. Что представляет это число, если не указано его назначение? В отличие от писателя хороший программист не должен рассчитывать на воображение пользователя.

ВОПРОСЫ

1. Что будет напечатано в результате выполнения следующей последовательности операторов?

```
WRITELN ('Важно ');
WRITE ('не ');
WRITE ('путать WRITE');
WRITE (' ');
WRITE ('и WRITELN ');
WRITELN ('Их действия ');
WRITE ('несколько различны.');
```

2. Напишите программу, которая будет считывать месяц, день и год. Программа должна печатать эти данные в стандартном формате мм — дд — гг. Например, при вводе

9 3 1961

должно быть выведено

9-3-61

*3. Преобразуйте программу в примере 2.1 так, чтобы ее можно было использовать для нахождения суммы различных слагаемых. Обеспечьте вывод результатов в удобной и понятной для пользователя форме.

4. Гипотетический пакет акций, оцениваемый в 10 000 долл., делится на пять равных частей. Напишите программу вычисления стоимости каждой части пакета акций. Программа должна выводить величину стоимости при заданном прогнозе процентного изменения курса каждой акции.

# Глава 4
# Встроенные типы данных

## 4.1. Тип данных REAL (вещественный)

Все предыдущие программы использовали переменные типа INTEGER, т. е. в программах решались только задачи целочисленной арифметики. Однако в большинстве применений ЭВМ требуется обработка нецелочисленных данных. В качестве примера задач, где требуются вещественные числа, можно рассматривать даже простые задачи по учету биржевых операций или научные расчеты, в особенности использующие логарифмические или тригонометрические функции.

В языке Паскаль существует встроенный тип данных REAL. Можно объявить, что переменная имеет тип REAL по аналогии с тем, как это делается для целочисленной переменной. В арифметике вещественных чисел используются те же операторы, что и в целочисленной арифметике, за исключением символа деления: зарезервированное слово *div* заменяется символом / . В отличие от целочисленного деления, которое вычисляет только целую часть частного, вещественное деление получает все частное полностью. Так,

5 div 2 = 2
*но* 5/2 = 2.5.

Определены также некоторые из упоминавшихся ранее функций. Функции (гл. 2) ABS(x) и SQR(x) можно использовать с вещественными аргументами. Помимо их с вещественными и целыми аргументами можно использовать следующие функции (они дают вещественный результат):

SIN(X) — синус X, заданного в радианах
COS (X) — косинус X, заданного в радианах
EXP (X) — $e^x$
LN (X) — натуральный логарифм X (при $X > 0$)
LOG (X) — логарифм X по основанию 10 (при $X > 0$)
SQRT (X) — квадратный корень из X (при $X \geqslant 0$)

ARCTAN (X) — арктангенс X, заданного в радианах
ATAN (X) — то же, что и ARCTAN

Функция PWROFTEN(X) вычисляет $10^X$ в форме вещественного числа, однако работает только с целым аргументом. Таким образом, PWROFTEN(4) — правильная запись, а PWROFTEN(3.5) — неверная запись функции.

Эти функции могут быть использованы в любом арифметическом выражении. Их аргументы в свою очередь могут быть выражениями. Это означает, что, так же как и в случае с выражениями типа INTEGER, программист может сконструировать любое вещественное выражение.

Вещественные числа могут быть записаны двумя способами. Первый — (естественный) способ состоит в том, что вещественные числа записываются в виде последовательности цифр с десятичной точкой. По обеим сторонам от точки должна располагаться по крайней мере одна цифра. Несколько примеров вещественных чисел при этом способе записи выглядят так:

3.1415, –0.5, +52.0, 1.0

Второй способ представления вещественных чисел — экспоненциальная форма, которая представляет собой вариант записи чисел, принятый в научных статьях. Например, число 900.1 может быть записано в виде

9.001E + 2

В экспоненциальной форме E означает «десять в степени». Число справа от E представляет показатель степени. (Необходимо заметить, что заглавная буква E не имеет ничего общего с математическим символом *e*.) Примеры:

5.00000E − 01 = $5 * 10^{-1} = 0.5$
5.00000E + 00 = $5 * 10^{0} = 5.0$
5.00000E + 01 = $5 * 10^{1} = 50.0$
6.37104E + 03 = $6.37104 * 10^{3} = 6371.04$
7.0E − 02 = $7.0 * 10^{-2} = 0.07$

Экспоненциальная форма удобна при записи очень больших или очень маленьких чисел. Например, часто встречающийся в начальных главах химии коэффициент преобразования, называемый *числом Авогадро*, может быть

представлен в операторе присваивания следующим образом:

```
ATOMS := MOLES * 6.02E + 23
```

**Пример 4.1. Вычисление синуса и косинуса**

```
program TRIG;
var
      ANGLE : REAL;
      RESULT1, RESULT2 : REAL;
begin
      WRITELN ('Задайте значение угла в радианах');
      READ (ANGLE);
      RESULT1 := SIN (ANGLE);
      RESULT2 := COS (ANGLE);
      WRITELN ('Синус равен', RESULT1 : 6 : 3);
      WRITELN ('Косинус равен', RESULT2 : 6 : 3)
end.
```

В программе примера 4.1 описывается три переменных: ANGLES, RESULT1 и RESULT2. В завершающих программу операторах WRITELN показано, каким образом можно определить размеры поля для представления вещественных чисел при печати. Спецификация размеров поля для представления вещественных чисел записывается в виде *:w:d*, где *w* показывает общий размер поля, а *d* — число позиций для цифр справа от десятичной точки. Если в примере задачи значение угла в радианах 1.047, т. е. $\pi/3$, в результате будет напечатано:

```
Синус равен 0.866
Косинус равен 0.500
```

Если не указано *d* или не определены размеры поля, результаты будут напечатаны в экспоненциальной форме:

```
Синус равен 8.65927E–01
Косинус равен 5.00000E–01
```

В языке Паскаль при обработке вещественных чисел учитываются первые шесть значащих цифр[1]. Так, число

[1] Имеется в виду конкретная реализация. В общем случае число десятичных цифр в представлении вещественного числа существенно зависит от ЭВМ и реализации. Например, в ЭВМ CDC 6600 используется 14 цифр. — *Прим. ред.*

1.234567 обычно округляется до 1.23457. Точно так же число 7.654321E+10 будет округлено до 7.65432E+10. Эта разница может показаться несущественной, однако в некоторых программах из-за эффекта накопления даже небольшие погрешности могут привести к значительным ошибкам вычислений.

Ошибки, возникающие из-за накопления погрешности округления, могут привести к неожиданным результатам. Хотя такое происходит не в каждой программе, тем не менее следует всегда быть внимательным. Например, нельзя быть уверенным в истинности простого равенства «X = 3 * (X/3)». Приведем пример вычисления значения выражения для двух значений X:

для X = 3.0, 3 * (X/3) = 3 * 1.00000 = 3.0
*но* для X = 1.0, 3 * (X/3) = 3 * 0.33333 = 0.99999

К счастью, эта ошибка не столь серьезна, как кажется. Так, при работе оператора вывода с соответствующими размерами поля неточная величина будет автоматически округляться перед печатью. Например, если X равен 0.99999, оператор WRITELN (X : 4 : 2) в действительности напечатает 1.00.

Некоторые реализации языка Паскаль вычисляют вещественные числа с большей точностью, чем 6 десятичных цифр. Когда пишется программа, использующая тип REAL, точность вычислений должна соответствовать точности вычислений на выбранном типе вычислительной машины. Необходимо также помнить, что результат вычисления трансцендентных функций (таких, как SIN или LN) обычно получается с меньшим числом точных цифр, чем результат других операций.

### 4.2. Взаимные преобразования чисел

Если переменной типа REAL присваивается значение типа INTEGER, то целое значение автоматически преобразуется в вещественный эквивалент. Однако переменной типа INTEGER нельзя присваивать вещественное значение. Таким образом, если COUNT — переменная типа INTEGER, то следующие операторы присваивания вызовут ошибки вида «несоответствие типов»:

```
COUNT := 3.14
COUNT := SQRT(2)
COUNT := 0.5
```

Ясно, что переменная типа INTEGER не может принять значения 0,5, так как 0,5 не имеет соответствующего целочисленного эквивалента. Поэтому соответствующий оператор присваивания вызовет ошибку при выполнении программы (конечно, если COUNT описана как переменная типа REAL, то оператор присваивания приемлем).

Чтобы обойти это ограничение, можно использовать имеющиеся в языке Паскаль функции *преобразования типов.* Эти функции используют в качестве аргументов переменные типа REAL и преобразуют их в целочисленные значения:

TRUNC(x) результат — целая часть x.
ROUND(x) результат — значение[1], округленное до ближайшего целого.

Различны ли эти функции? В некоторых случаях они дают одинаковые значения. При x = 3.2 и TRUNC, и ROUND дадут значение 3. Но при x = 3.6 функция TRUNC даст значение 3, в то время как ROUND возвращает значение 4. Это объясняется тем, что функция ROUND анализирует дробную часть аргумента, чтобы решить, округлять x до ближайшего большего или до ближайшего меньшего целого значения, в то время как TRUNC просто отбрасывает дробную часть.

Таким образом,

TRUNC (5.8) → 5
ROUND (5.8) → 6
TRUNC (3.14) → 3
ROUND (3.14) → 3
TRUNC (–7.7) → –7
ROUND (–7.7) → –8

Целые и вещественные переменные можно использовать одновременно в одном арифметическом выражении. Если некоторый операнд выражения имеет тип REAL,

---

[1] Стандартную функцию ROUND можно выразить через стандартную функцию TRUNC следующим образом:

$$\text{ROUND}(x) = \begin{cases} \text{TRUNC}\ (x + 0{,}5), & x \geqslant 0; \\ \text{TRUNC}\ (x - 0{,}5), & x < 0. \end{cases}$$

— *Прим. ред.*

то и все выражение будет иметь тип REAL. Нужно помнить, что вещественные функции, подобные SQRT(x), перечисленные в разд. 4.1, всегда возвращают вещественный результат. Это означает, что функция SQRT(4) возвращает вещественное значение 2.0, а не целое 2. Прежде чем это значение будет присвоено целочисленной переменной, оно должно быть преобразовано в целое с помощью одной из функций преобразования типов.

## 4.3. Тип данных BOOLEAN

Не всегда переменные используются для того, чтобы представлять числа. Одним из нечисловых типов данных является тип данных BOOLEAN. Булевы (логические) переменные могут иметь одно из двух значений: TRUE (истина) или FALSE (ложь). По сравнению с типом REAL, который допускает миллионы различимых значений, может показаться, что тип BOOLEAN имеет ограниченную область применения. Однако, как это ни странно, именно ограниченность диапазона значений булевых переменных придает булевым выражениям их мощь.

Булевы выражения могут принимать несколько различных форм. Во-первых, они могут быть просто константами TRUE (истина) или FALSE (ложь). Оператор присваивания, использующий эту форму, аналогичен арифметическому оператору вида "ADDEND1 := 3". Покажем два оператора, которые присваивают булевы значения булевым переменным:

```
FLAG := TRUE
VALID := FALSE
```

**Отношения**

| | |
|---|---|
| = | равно |
| > | больше, чем... |
| >= | больше, чем... или равно |
| < | меньше, чем... |
| <= | меньше, чем... или равно |
| <> | не равно |

Булевы выражения можно также использовать, чтобы проверить отношение между двумя переменными. Допустим, у нас есть две переменные типа INTEGER и нужно узнать, равны ли они. Если имена перемен-

ных FIRST и SECOND, мы можем проверить их равенство друг другу с помощью булевого выражения

FIRST = SECOND

Если они равны, это выражение примет значение TRUE. Если они не равны, выражение примет значение FALSE. Отметим, что знак равенства (=) отличен от знака присваивания (:=). Другие знаки отношений приведены выше.

Кроме того, булевы выражения могут конструироваться с помощью булевых операций. Эти операции образуют инструментальный фундамент булевой логики, алгебры логики, разработанной в XIX в. математиком Джорджем Булем. В язык Паскаль включены три булевы операции: *and* (и), *or* (или) и *not* (не).

Результат операции *and* есть истина, только если оба ее операнда истинны. Так,

TRUE and TRUE → TRUE
FALSE and FALSE → FALSE
TRUE and FALSE → FALSE
FALSE and TRUE → FALSE

Результат операции *or* есть истина, если какой-либо из ее операндов истинен. Так,

TRUE or TRUE → TRUE
FALSE or FALSE → FALSE
TRUE or FALSE → TRUE
FALSE or TRUE → TRUE

Третья операция *not* имеет один операнд и образует его логическое отрицание. Результат операции *not* есть FALSE, если операнд истинен, и TRUE, если операнд имеет значение ложь. Так,

not TRUE → FALSE
not FALSE → TRUE

Когда полезны эти операции? Если нужно проверить сложное условие. Допустим, имеются четыре вещественные переменные: HOURS1, HOURS2, WAGES1 и WAGES2. Будем проверять различные условия и помещать результаты проверки в качестве значения переменной DUMMY.

Если нужно определить, равны ли HOURS1 и HOURS2, можно просто написать

DUMMY := (HOURS1 = HOURS2)

DUMMY теперь содержит булев результат сравнения. Если нужно не только знать, равны ли HOURS1 и HOURS2, но также равны ли при этом WAGES1 и WAGES2, можно использовать операцию *and:*

```
DUMMY := (HOURS1 = HOURS2)  and  (WAGES1 = WAGES2)
```

Если оба отношения истинны, то и результат операции *and* — истина. Если какое-либо отношение ложно, то и результат операции *and* — ложь. Подобную проверку можно конструировать, для того чтобы определить наличие хотя бы одного равенства

```
DUMMY := (HOURS1 = HOURS2)  or  (WAGES1 = WAGES2)
```

Далее в программе можем присвоить DUMMY значения его логического отрицания:

```
DUMMY := not DUMMY
```

Булевы выражения, так же как и в случае целых или вещественных выражений, могут быть сложными. Они могут использовать три булевы операции в любых сочетаниях; операнды могут быть простыми булевыми константами, булевыми переменными или отношениями. Если для указания порядка выполнения действий используются скобки, можно не знать старшинства операций в булевом выражении, в противном случае его нужно учитывать.

---

**Порядок старшинства операций в булевых выражениях**

*Высший* ( )
not
and
or
*Низший* > = <> >= <=
*Примеры:*
I and J or K → (I and J) or K
not X and Y → (not X) and Y

---

В языке Паскаль имеется встроенная булева функция ODD(x) с целочисленным аргументом. Если целое выражение x нечетно, ODD(x) принимает значение истина; если x четно, тогда ODD(x) ложно. Можно

провести аналогию с использованием операции mod. Например, оператор присваивания

```
DUMMY := ODD (COUNT)
```

эквивалентен оператору присваивания

```
DUMMY := (COUNT mod 2) = 1
```

### 4.4. Описание констант

В работе программ возможна такая ситуация, когда одна и та же числовая величина используется несколько раз. Например, в программах, выполняющих тригонометрические преобразования, часто встречается число 3.14159. Если это так, то такому числу удобно присвоить имя — описать его как константу. Это сделано в программе примера 4.2. В описании константы идентификатор имени слева от знака равенства является символическим представлением константы, расположенной справа от знака равенства. Далее в программе этот идентификатор используется так, как если бы это была соответствующая константа. Значение справа от знака равенства должно быть простой константой; оно не может быть выражением. Более того, константе не может быть присвоено новое значение, после того как она объявлена.

---

**Пример 4.2. Площадь и длина окружности**

```
program CIRCLE;
const
      PI = 3.14159;
var
      RADIUS : REAL;
begin
      WRITELN ('Каков радиус круга?');
      READ(RADIUS);
      WRITELN ('Длина окружности =', (2 * PI * RADIUS)
                                                  : 6 : 2);
      WRITELN ('Площадь круга =', (PI * SQR (RADIUS))
                                                  : 6 : 2)
end.
```

---

Раздел описания констант всегда предшествует разделу описания переменных. В отличие от описания переменных описание констант не специфицирует используемый тип данных; машина может это определить по виду константы. Заметим, что правила выбора идентификаторов для констант те же, что и в случае переменных. Приведем примеры правильного описания констант:

```
FREEZING_POINT = 0
SPEED_LIMIT = 55
DAYS_PER_YEAR = 365.25
SPEED_OF_LIGHT = 2.998E + 10
DEVICE_WORKING = TRUE
```

При описании константы ей желательно присваивать осмысленное имя, тогда тем, кому в дальнейшем потребуется разобраться в работе этой программы, будет существенно легче. Во-первых, символические константы делают программы понятнее. Читающий чужую программу будет только благодарен ее автору за использование идентификатора, например WEEKS_PER_YEAR (число недель в году), вместо часто повторяющегося необъясненного числа 52. Несомненно, тот, кто разбирается в чужой программе, в конце концов поймет, что́ представляет собой некоторое числовое значение, однако и его время, и его умственную энергию можно сохранить, если замысел автора выразить явно с помощью описания констант. (Даже если автор — единственный пользователь своей программы, нужно придерживаться хорошего стиля программирования. Небрежно написанные программы могут ввести в заблуждение и самого автора).

Во-вторых, использование символических констант облегчает модификацию программ. Если нужно изменить некоторое часто используемое числовое значение, проще изменить только одну строку описания константы, чем просматривать всю программу и вносить исправление при каждом появлении этого значения. Допустим, нужно написать программу, которая производит некоторые вычисления и затем строит последовательность точек на терминале ввода-вывода. Для того чтобы правильно масштабировать точки, программа должна знать, сколько позиций помещается на строке. Существующие в настоящее время терминалы

в зависимости от типа имеют различную длину строки: на одних можно напечатать 132 знака в строке, в то время как другие допускают только 80 или 72.

Конечно, возможен такой подход к написанию программы, когда длина строки явно указывается в каждом использующем ее выражении. Однако если позже будет получен новый терминал или если программу передадут на ЭВМ, оснащенную другим типом терминала, то каждое появление значения длины строки должно быть изменено соответствующим образом. Лучший подход состоит в описании константы

```
LINELENGTH = 80
```

Теперь можно изменить нужную величину, переписав только одну строчку. Такое решение делает программу более удобной для чтения и эксплуатации. LINELENGTH можно использовать в выражениях подобно переменным:

```
SCALE := MAXVALUE / LINELENGTH
```

Ее можно использовать при указании размера поля:

```
WRITELN (TOTAL : LINELENGTH)
```

Эти операторы выполняют те же действия, как и в случае, если символическое имя LINELENGTH заменить значением константы 80. Если позже программу изменят так, что для LINELENGTH будет установлено значение 72, LINELENGTH будет восприниматься как 72.

Язык Паскаль содержит встроенную константу MAXINT. Значение MAXINT показывает величину самого большого числа, представимого типом INTEGER. MAXINT равен $2^{15}-1$ или 32767[1]. Следовательно, все значения типа INTEGER должны лежать в диапазоне от –32767 до +32767. Присваивание целочисленной переменной значения за пределами этого диапазона вызовет ошибку переполнения.

---

[1] Имеется в виду конкретная реализация. Более того, существуют реализации, где помимо MAXINT используется константа MININT, причем MININT ≠ – MAXINT. — *Прим. ред.*

ВОПРОСЫ

*1. Переменные I и J описаны как INTEGER; переменные X и Y описаны как REAL. Какой из следующих операторов присваивания вызовет ошибку «несоответствие типов»?

```
X := SQRT(Y)
X := SQRT(I)
Y := SIN(J)
I := SIN(X)
J := COS(I)
X := TRUNC(Y)
J := X / I
I := J * ROUND(X)
```

*2. Что́ будет напечатано при выполнении следующих ниже операторов, если SUBTOTAL равен 17.693?

```
WRITELN (SUBTOTAL : 7 : 3)
WRITELN (SUBTOTAL : 7 : 2)
WRITELH (SUBTOTAL : 7 : 1)
WRITELN (SUBTOTAL : 5 : 1)
WRITELN (SUBTOTAL : 10)
WRITELN (SUBTOTAL)
```

*3. Даны четыре булевы переменные, которые содержат следующую информацию о людях:

| | |
|---|---|
| MARRIED | истинна, если человек женат (замужем); |
| BLOND | истинна, если у человека светлые волосы; |
| MALE | истинна, если человек мужчина; |
| EMPLOYED | истинна, если человек работает. |

Напишите булевы выражения, с помощью которых определяется, является ли человек:

а) замужней женщиной;
б) неженатым мужчиной;
в) незамужней блондинкой;
г) безработной незамужней женщиной;
д) либо неженатым, либо безработным, либо и тем и другим вместе.

4. Напишите программу, которая вводит два вещественных числа, затем печатает первое число, возведенное в степень второго числа. Для возведения в степень вещественных чисел в языке Паскаль используйте следующее тождество:

$$x^y = e^{y \ln x}$$

# Глава 5
# Условный оператор

## 5.1. Ветвление

Используя булевы выражения, можно осуществлять проверку различных условий. Можно, например, найти булев результат сравнения. Язык Паскаль содержит условный оператор *if...then...else,* который позволяет использовать результат вычисления булева выражения для выбора пути выполнения программы.

Проверяемое условие может быть любым булевым выражением. Этот оператор называют *оператором ветвления,* так как он позволяет в зависимости от результата проверки условия переходить на ту или иную ветвь программы. После того как действие, определенное оператором *if,* выполнено, программа продолжает выполняться со следующего за *if* оператора.

Программа из примера 5.1 решает уравнение $Ax^2 + Bx + C = 0$; A, B и C представлены переменными QUAD, LINEAR и CON соответственно. После их ввода программа определяет число вещественных корней уравнения. Так как условия TEMP $< 0$, TEMP $= 0$ и TEMP $> 0$ взаимно исключают друг друга, можно переписать программу в следующем виде:

```
if TEMP < 0 then
    WRITELN ('У этого уравнения нет вещественных корней.')
else
    if TEMP = 0 then
        WRITELN ('У этого уравнения один вещественный корень.')
    else
        WRITELN ('У этого уравнения два вещественных корня.')
```

IF ... THEN ... ELSE

Общая форма записи:

if *булево выражение* then *оператор*

if *булево выражение* then *оператор*$_1$

else *оператор*$_2$

В первой форме оператор выполняется, только если *булево выражение* истинно. Во второй форме *оператор*$_1$ выполняется, если *булево выражение* истинно; если оно ложно, то выполняется *оператор*$_2$.

Примеры:

```
if  TEMP > 0 then FREEZING : = FALSE
if  MARRIED then
        WRITELN ('Человек женат.')
if  I = J  then
        WRITELN ('Числа равны.')
else
       WRITELN ('Числа не равны.')
```

**Пример 5.1. Корни квадратного уравнения**

```
program QUADRATIC;
var
      QUAD, LINEAR, CON : INTEGER;
      TEMP : INTEGER;
begin
     WRITELN ('Пожалуйста, введите коэффициенты
                                          при');
      WRITELN ('Квадратном, линейном и постоянном
                                          членах:');
      READ (QUAD, LINEAR, CON);
      TEMP : = SQR (LINEAR) - 4 * QUAD * CON;
      if TEMP < 0 then
      WRITELN ('Уравнение не имеет вещественных
                                          корней!');
      if TEMP = 0 then
        WRITELN ('Уравнение имеет один вещественный
                                          корень!');
      if TEMP > 0 then
        WRITELN ('У уравнения два вещественных кор-
                                          ня!')
end.
```

Когда оператор *if* появляется внутри другого оператора *if*, они считаются *вложенными*. Такое «вложение» используется для уменьшения числа необходимых проверок. Этот метод часто обеспечивает большую эффективность, однако одновременно он уменьшает наглядность программы. Не рекомендуется использовать более одного-двух уровней вложения *if*. За вторым уровнем вложения становится трудно восстановить последовательность проверки условий каждым *if*.

Если часть *else* используется во вложенных *if*, то каждое *else* соответствует тому *if*, которое ему непосредственно предшествует. Таким образом, при определении последовательности выполнения фрагментов нет двусмысленности. Первый WRITELN выполняется, если TEMP < 0, второй — если TEMP = 0, третий — если TEMP > 0.

## 5.2. Составной оператор

Управляющая структура *if* может показаться негибкой, так как выполняемые действия могут быть описаны только одним оператором. Иногда может потребоваться выполнение последовательности операторов. В этом случае хотелось бы заключить всю последовательность в воображаемые скобки, например:

```
if AMOUNTDUE > CREDITLIMIT then
        WRITELN ('Этот поставщик превысил предельный кредит.')
        ACCEPTORDER : = FALSE;
```

Когда необходимо добиться того, чтобы последовательность операторов работала как единый оператор, можно помещать эту последовательность между зарезервированными словами *begin* и *and*. Предыдущий оператор *if* может быть переписан следующим образом:

```
if AMOUNTDUE > CREDITLIMIT then
  begin
        WRITELN ('Этот поставщик превысил предельный кредит.');
        ACCEPTORDER : = FALSE
  end;
```

Конструкция *begin...and,* которую мы здесь сформировали, называется *составным оператором.* Составной оператор часто используется в условных операторах, так что выполняемые действия могут включать в себя более одного оператора. Программа примера 5.2 использует составной оператор во вложенном операторе *if.*

---

**Пример 5.2. Квадратное уравнение**

```
program QUADFORMULA;
var
      QUAD, LINEAR, CON : REAL;
      RADICAL, ROOT1, ROOT2 : REAL;
begin
      WRITELN ('Введите, пожалуйста, коэффициенты');
      WRITELN ('и константу квадратного уравнения:');
      READ (QUAD, LINEAR, CON);
      RADICAL : = SQR (LINEAR) - 4 * QUAD * CON;
      if RADICAL < 0 then
         WRITELN ('Уравнение не имеет вещественных
                                              корней.')
      else
         if RADICAL = 0 then
              WRITELN ('У уравнения корень',
            -LINEAR / (2 * QUAD))
         else
              begin
                ROOT1 : = (-LINEAR + SQRT
                              (RADICAL)) / (2 * QUAD);
                ROOT2 : = (-LINEAR - SQRT)
                              (RADICAL)) / (2 * QUAD);
                WRITELN ('У уравнения два корня: ',
                                              ROOT1);
                WRITELN ('и ', ROOT2)
              end
end.
```

---

Отметим, что не каждый оператор в этом примере заканчивается точкой с запятой. Хотя большинство операторов в программах на языке Паскаль заканчиваются точкой с запятой, после некоторых операторов точка с запятой не ставится. В гл. 2 уже упоминалось одно исключение: зарезервированные слова

*begin* и *and*. Правила употребления точки с запятой несколько условны, но согласованны. Необходимо помнить три принципа:

1. Каждое описание переменной и определение константы заканчиваются точкой с запятой.
2. Каждый оператор в теле программы завершается точкой с запятой, если сразу вслед за ним не следуют зарезервированные слова *end, else* или *until.*
3. После определенных зарезервированных слов, таких, как *then, else, var, const* и *begin,* никогда не ставится точка с запятой.

### 5.3. Организация циклов

*Цикл* — это последовательность операторов, которая может выполняться более одного раза. В языке Паскаль имеется несколько механизмов для конструирования циклов. Простейший из них — оператор *while.* При выполнении оператора *while* повторяется определенная группа операторов до тех пор, пока определенное в операторе *while* булево условие истинно.

---

**WHILE ... DO**

Общая форма записи:

while *булево выражение* do *оператор*

Указанный оператор (или составной оператор) выполняется повторно до тех пор, пока *булево выражение* истинно. Если оно ложно, когда while выполняется в первый раз, указанный оператор вообще не выполняется.

*Примеры:*

```
while DEGREES > 180 do
      DEGREES : = DEGREES - 360
while COUNT < 10 do
      begin
            READ (NUMBER);
           SUM : = SUM + NUMBER,
            COUNT : = COUNT + 1
      end
```

---

Программа, показанная в примере 5.3, находит среднее арифметическое последовательности чисел; размер последовательности — символическая константа. Каж-

дый раз, когда вводится число, COUNT увеличивается на 1. Таким образом, цикл *while* будет выполнен ровно столько раз, сколько указано значением LISTSIZE. Для того чтобы программа работала, перед входом в цикл переменным SUM и COUNT должны быть присвоены нулевые начальные значения.

---

**Пример 5.3. Нахождение среднего арифметического последовательности чисел**

```
program AVERAGE3;
const
    LISTSIZE = 10;
var
    SUM, NUMBER : REAL;
    COUNT : INTEGER;
begin
    SUM := 0;
    COUNT : = 0;
    WRITELN ('Пожалуйста, введите ваш',
                    LISTSIZE : 1,' — элементный список.');
    while COUNT < LISTSIZE do
        begin
            READ (NIMBER);
            SUM : = SUM + NUMBER;
            COUNT : = COUNT + 1
        end;
    WRITELN ('Среднее — ', SUM / COUNT)
end.
```

---

Программа будет более гибкой, если ее можно будет использовать для последовательностей различной длины. Один из способов заключается в том, чтобы ввести значение LISTSIZE с помощью оператора READ. У этого подхода один недостаток: пользователь должен подсчитывать число элементов в последовательности.

Альтернативное решение состоит в том, чтобы исключить из употребления одно значение и использовать его как стоп-код. Если мы заранее знаем, что −1 никогда не появится в последовательности, то можно использовать −1 для указания ее конца. Используя такой способ, необходимо перед входом

в цикл *while* прочитать первое значение NUMBER, иначе NUMBER было бы не определено при первой проверке условия *while*. В примере 5.4 показано, как могла бы выглядеть измененная программа.

---

**Пример 5.4. Прекращение ввода с помощью стоп-кода.**

```
program AVERAGE4;
const
    STOPCODE = -1;
var
    SUM, NUMBER : REAL;
    COUNT : INTEGER;
begin
    SUM : = 0;
    COUNT : = 0;
    WRITELN ('Пожалуйста, введите список');
    WRITELN ('заканчивающийся ', STOPCODE : 1);
    READ (NUMBER);
    while NUMBER <> STOPCODE do
        begin
            SUM : = SUM + NUMBER;
            COUNT : = COUNT + 1;
            READ (NUMBER)
        end;
    WRITELN ('Среднее значение равно ', SUM /
                                        COUNT)
end.
```

---

Теперь программа более гибкая, так как пользователь не должен заранее знать размер последовательности.

В языке Паскаль имеется булева функция EOF (INPUT), которая позволяет определить конец ввода без того, чтобы устанавливать специальное значение для стоп-кода. EOF соответствует ситуации «конец файла». Если не осталось никаких входных данных, то EOF принимает значение TRUE; пока же данные не исчерпаны, EOF имеет значение FALSE. Чтобы использовать EOF в программе нахождения среднего арифметического, нужно просто заменить оператор *while* следующим: while not EOF (INPUT) do

При вводе с терминала завершение последовательности должно быть указано нажатием клавиши «конец файла» (end-of-file). В большинстве систем этот знак изображается комбинацией CTRL − C″. Этот знак можно ввести, нажав одновременно клавишу «CTRL» и клавишу "C". Когда этот знак будет прочитан, функция EOF(INPUT) принимает значение TRUE.

Часто причина неправильной работы программы заключается в неправильном проектировании цикла. Одной из наиболее распространенных причин неверной работы программы является бесконечный цикл. *Бесконечный цикл* — это такой цикл, который никогда не прекращается. Обычно он возникает из-за ошибок в логике программы, а не из-за синтаксических ошибок. В приведенном ниже примере цикл будет выполняться неопределенно долго, так как программист забыл увеличить COUNT:

```
SUM : = 0;
COUNT : = 0;
while COUNT < LISTSIZE do
      begin
          READ (NUMBER);
          SUM : = SUM + NUMBER
      end;
```

Бесконечные циклы возникают и тогда, когда цикл сконструирован таким образом, что проверяемое условие никогда не становится ложным. Иногда такие бесконечные циклы возникают из-за простой небрежности, как в предыдущем примере. Однако чаще они являются следствием неверного понимания программистом работы программы.

ВОПРОСЫ

1. Предположим, что DUMMY присвоено некоторое значение. Чтó произойдет при выполнении следующих операторов?

```
while DUMMY = DUMMY do
      WRITELN (' * ')
```

2. В приведенном ниже операторе *if* возникнет арифметическая ошибка при $X \leqslant 0$. Используя вложенные *if*, перепишите оператор так, чтобы LN(X) не вычислялся при $X \leqslant 0$.

```
if (X > 0) and (LN(X) >= 0) then
    WRITELN ('X в степени Y равен ', EXP(Y * LN(X)) : 5)
```

3. В каком случае приведенная ниже последовательность операторов приведет к ошибке переполнения?

```
COUNT : = 0;
while COUNT <= MAXINT do
    COUNT := COUNT + 1
```

4. Напишите программу, которая обрабатывает значения температуры (в градусах Цельсия), поступающие от пользователя. Для каждого значения программа должна определить, является ли данная температура меньшей, равной или большей, чем точка замерзания воды. После того как все значения прочитаны, программа должна напечатать их общее число в списке, а также наибольшее и наименьшее значение температуры.

*5. Напишите программу, которая вводит целое число и определяет, является ли оно простым числом.

# Глава 6
# Являетесь ли вы хорошим программистом?

## 6.1. На кого следует ориентироваться при разработке программы

Часто ли нам приходится писать другим людям? Вероятно, достаточно часто. Практически все, что пишется — записки, письма, научные статьи и т. д., — предназначено для других людей.

Когда вы пишете, то, вероятно, стараетесь строить изложение просто и в соответствии с правилами грамматики не столько из-за необходимости следовать этим правилам, сколько из желания достичь легкости восприятия написанного. Другими словами, хороший стиль изложения помогает вам быстрее достичь понимания. Бессвязное и неаккуратное написание свидетельствует об отсутствии заботы о читателе.

Одна из фаз разработки программ связана с ее написанием. Помимо программиста-разработчика большинство программ читается и используется другими людьми. Следовательно, программист, который пишет запутанно, не соблюдает дисциплину программирования, подобен автору витиеватых фраз.

Стиль программирования — это не просто набор правил, которым надо слепо следовать. Это прежде всего выражение опыта общения. При написании программы следует учитывать мнение трех лиц. Прежде всего мнение человека, который будет использовать программу. Пользователь программы, подобно читателю, хочет, чтобы информация представлялась в ясной и легко усваиваемой форме. Как отмечалось в гл. 3, неразумно — и не следует — требовать от пользователя понимания плохо построенного вывода.

В данном контексте термин «вывод» означает не только представление окончательных результатов, но и подразумевает все промежуточные выдачи, которые происходят по ходу работы программы. Одна из важных функций промежуточной выдачи заключается в том, чтобы попросить пользователя ввести данные. Сообщения, которые программа использует для такого

запроса, называются *побуждающими сообщениями.* Следует внимательно подходить к составлению побуждающих сообщений, как, впрочем, и любого вывода для уменьшения вероятности их неправильного толкования.

Вторым лицом, мнение которого необходимо учитывать, является программист, который захочет ознакомиться с программой. Желание прочитать чужую программу может возникнуть по ряду причин, например для того, чтобы использовать в своей программе алгоритм, разработанный другим программистом, или чтобы адаптировать чужую программу к своим требованиям. В любом случае автору, написавшему понятную и хорошо составленную программу, будут благодарны.

Сам автор является третьим лицом, которое должно приниматься во внимание при разработке программы. Даже если никто не будет использовать или проверять написанную программу, усилия, направленные на достижение легкости в понимании программы, будут потрачены ненапрасно. При разработке более сложных программ уже не придется преодолевать трудности проектирования. Следует с самого начала приучиться к хорошему стилю программирования и в дальнейшем просто следовать этим элементам проектирования.

## 6.2. Что рекомендуется и что не рекомендуется хорошим программистам

Хотя методика программирования в значительной степени строится на здравом смысле, некоторые ее элементы настолько существенны, что требуют формализации. Советы, что следует делать и чего следует избегать, не преследуют цель поставить программиста в жесткие рамки. Эти правила помогут писать более совершенные программы. Если, исходя из здравого смысла, постоянно выполнять эти рекомендации, то программы станет легче писать и использовать.

● *Рекомендуется использовать значащие идентификаторы.* Если используется переменная, которая содержит значение полной выплаты (GROSS PAY), то ей не следует присваивать имя XI или ABC, а следует обозначать ее GROSSPAY. Программист, как и любой

автор, должен обеспечить контакт с пользователем. Лучшим способом передать смысл переменной или символической константы является присвоение ей имени, которое отражает назначение переменной или константы.

• *Рекомендуется использовать ступенчатую форму записи программы.* В данной книге примеры программ построены с использованием сдвигов таким образом, что легко распознается каждый уровень вложения. Оператор (или составной оператор) в конструкции *if* выделяется несколькими пробелами. Так же выделяется внутренний оператор в операторе цикла *while.* На ряде машин такого рода редактирование текста программы выполняется автоматически программами так называемой *форматированной печати.* Если нельзя воспользоваться услугами такой программы, то старайтесь пользоваться правилами сдвига текста, применяемыми в примерах программ.

• *Рекомендуется писать побуждающие сообщения ясным языком.* При составлении побуждающих сообщений не следует стремиться к литературным шедеврам, вполне достаточно однозначного понимания этих сообщений. К сожалению, у некоторых программистов чувствуется тяготение к кричащим побуждающим сообщениям с загадочными сокращениями и тяжелой структурой. Идеальное обращение является ясным и не отпугивает пользователя. Короче говоря, оно должно быть построено таким образом, чтобы обеспечить быстрое понимание.

• *Не рекомендуется использовать одну и ту же переменную для нескольких целей.* Описав переменную, можно поддаться соблазну получить дополнительные выгоды от ее использования для новых целей в другой части программы. Например, завершив использование переменной GROSSPAY для хранения полной выплаты, может возникнуть желание в дальнейшем использовать эту переменную в качестве счетчика цикла. Следует ли пользоваться такой экономией?

Конечно, нет. Все, что удается сэкономить при повторном использовании GROSSPAY, — это незначительные затраты на описание новой переменной. К тому же такое сокращение сопровождается значительным уменьшением наглядности программы. Что хорошего, если целенаправленное имя, подобное

GROSSPAY, используется для маскировки счетчика цикла? При написании программы еще имеется возможность помнить такую двойственность в использовании переменной GROSSPAY. Но если в будущем возникнет необходимость модифицировать эту программу, то вновь придется столкнуться с бесполезной тратой времени на выяснение целей использования переменной в каждом случае.

• *Не рекомендуется размещать несколько операторов на одной строке.* В языке Паскаль допускается размещение на одной строке нескольких операторов. Однако следует избегать такой практики, так как это затрудняет понимание структуры программы. В примерах 6.1 и 5.3 записана одна и та же программа.

---

**Пример 6.1. Программа, которую трудно читать**

```
program SQUISHED (INPUT, OUTPUT); const LISTSIZE = 10; var SUM, NUMBER : REAL; COUNT : INTEGER; begin SUM : = 0; COUNT := 0; WRITELN ('Пожалуйста, введите ваш ',LISTSIZE : 3, ' — элементный список.'); while COUNT < LISTSIZE do begin READ (NUMBER); SUM := SUM + NUMBER; COUNT := COUNT + 1 end; WRITELN ('Среднее =', SUM / COUNT) end.
```

---

Признайтесь, что теперь эту программу читать намного труднее. По сравнению с предыдущим текстом программы, в которой для выделения логики вложения используют сдвиги, в сжатой версии это на обеспечивается. При размещении нескольких операторов на одной строке создаются искусственные трудности для чтения такой программы.

• *Не рекомендуется гоняться за эффективностью.* Время от времени в журналах, статьях, учебниках по вычислительной технике встречаются призывы к эффективности. Эти советы непременно сводятся к выигрышу нескольких миллисекунд за счет сомнительных уловок. Эти маленькие хитрости незначительно влияют на эффективность, но влекут за собой существенные жертвы в наглядности и структурности программы.

Эффективность сама по себе вещь неплохая. К сожалению, поиски эффективного кодирования приводят к крайностям. Излишнее увлечение эффектив-

ностью программирования заслоняет более важную цель — легкость понимания программы.

### 6.3. Использование комментариев для пояснений

Одним из способов достижения легкости понимания программы является включение в программу комментариев. *Комментарий* — это текст, вставленный в программу для пояснения определенных аспектов кодирования. Вычислительная машина не воспринимает комментарии, и они не влияют на работу программы. Комментарии предназначены для более легкого понимания программы человеком, который с ней знакомится.

---

**Пример 6.2. Составной процент**

```
program COMPOUND;
(*Эта программа вычисляет доход по срочному вкладу
с учетом сложного годового процента.*)
var
AMOUNT : REAL;      (*Сумма денег*)
INTEREST : REAL;    (*Ежегодный рост прибыли*)
TIME : INTEGER;     (*Количество лет*)
begin
    (*-- Запрос величины основного капитала, процента и срока --*);
    WRITELN ('Какова величина основного капитала?');
    READ (AMOUNT);
    WRITELN ('Какова доля прироста в процентах?');
    READ (INTEREST);
    INTEREST := INTEREST / 100;   (*преобразование
    процентов к десятичному значению*)
    WRITELN ('За сколько лет выплачиваются проценты?');
    READ (TIME);
    (*-- Вычисление окончательного результата --*)
    while TIME > 0 do
        begin
            AMOUNT := AMOUNT * (1 + INTEREST);
            TIME := TIME - 1
        end;
    (*-- Выдача окончательного результата --*)
    WRITELN ('Окончательная величина', AMOUNT : 9
    : 2)
end.
```

---

В примере 6.2 демонстрируется использование комментариев. Все, что заключено между символами "(*" и "*)", трактуется как комментарий[1]. Отсутствуют какие-либо ограничения на длину и содержание комментария. В текст комментария могут включаться цифры, буквы и даже зарезервированные слова.

Комментарии могут появляться в любом месте программы. Однако запрещается вставлять комментарии в середину зарезервированного слова, идентификатора или числа. Если комментарий вставляется внутрь символьной строки, он будет рассматриваться как составная часть этой строки. Хотя размещение комментария внутри оператора не запрещается, однако следует избегать таких применений. Кроме опасности появления синтаксических ошибок комментарии, вставленные внутрь оператора, обычно затрудняют процесс чтения и понимания программы.

Комментарии, использованные в примере, могут быть разбиты на четыре класса. К первому классу относится комментарий, расположенный под заголовком, который объясняет назначение программы. Второй класс состоит из комментариев, в которых приводится назначение каждой константы и переменной. К третьему классу относятся комментарии, которые разбивают программу на ряд разделов из взаимосвязанных операторов. В четвертый класс включаются комментарии, объясняющие необычные или неочевидные аспекты используемого алгоритма.

Отсутствуют универсальные правила для определения мест, где следует располагать комментарии. Размещение и содержание комментариев в основном зависит от личных представлений программиста. В примере 6.2 показан образец хорошего комментирования, так как приведенные комментарии позволяют ответить на большинство возникающих вопросов. Что делает данная программа? Что каждая переменная обозначает? Какова цель данного оператора? Одним словом, достиг ли программист своих целей?

---

[1] В стандарте языка Паскаль комментарии заключаются в фигурные скобки {и}. В большинстве же реализаций, в том числе и в описываемой в данной книге, используются указанные комбинации символов. — *Прим. ред.*

# Глава 7
# Другие условные операторы

## 7.1. Еще о построении циклов

Как уже отмечалось, цикл можно образовать при помощи оператора *while*. При выполнении этого оператора ЭВМ вычисляет значение условия. Если условие истинно, то исполнительная часть оператора *while* будет выполняться до тех пор, пока это условие не примет значение FALSE. Если значение условия есть FALSE в самом начале, то исполнительная часть оператора *while* вообще не будет выполняться.

Что необходимо для создания цикла, который должен выполняться по крайней мере один раз? Для этого следует использовать оператор *repeat...until.*

Есть небольшое отличие в организации цикла *repeat* по сравнению с *while:* для выполнения в цикле *repeat* нескольких операторов не следует помещать эти операторы в скобки, состоящие из *begin* и *end.* Зарезервированные слова *repeat* и *until* действуют как составные операторные скобки. В примере 7.1 показана программа, в которой используется цикл *repeat.*

---

**REPEAT ... UNTIL**
Общая форма записи:
repeat
*Список операторов*
until *условие*
Конструкция *repeat ... until* работает аналогично циклу *while.* Различие заключается в том, что *while* проверяет условие до выполнения действий, в то время как *repeat* проверяет условие после выполнения действий. Это гарантирует хотя бы одно выполнение действий до завершения цикла:
*Примеры:*

```
repeat
    READ (NUMBER);
    SUM : = SUM + NUMBER
```

```
until NUMBER = -1
repeat
    COUNT : = COUNT + 1;
    WRITELN (SQR (COUNT))
until COUNT = 25
```

---

Эта программа определяет, является ли число простым, используя цикл *repeat*. При помощи операции *mod* проводится проверка всех целых чисел от 2 до NUMBER на принадлежность к множителям числа NUMBER. Если такой множитель находится, цикл завершается со значением COUNT, равным этому множителю. Если число является простым, цикл завершается при значении COUNT, равном NUMBER. (Цикл не может продолжаться бесконечно, так как любое число всегда делится само на себя.)

---

**Пример 7.1. Простые числа**

```
program PRIME;
(* Программа определяет, является ли число простым. *)
var
NUMBER : INTEGER;    (* Исследуемое число *)
COUNT : INTEGER;     (* Возможный делитель *)
begin
    (* -- Ввести исследуемое число -- *)
    WRITELN ('Какое число должно быть проверено?');
    READ (NUMBER);
    (* -- Число простое? -- *)
    COUNT : = 1;
    repeat
        COUNT : = COUNT + 1
    until NUMBER mod COUNT = 0;
    (* -- Вывести результаты -- *)
    if COUNT = NUMBER then
        WRITELN (NUMBER, '  является простым.')
    else
        WRITELN (NUMBER, ' делится на ', COUNT)
end.
```

---

В примере 7.2 иллюстрируется случай неверного использования оператора *repeat*. В чем заключается имеющаяся некорректность?

Эта программа, представляющая некоторую модификацию примера 5.4, не будет работать из-за плохой организации цикла. Когда значение STOPCODE встречается в списке, цикл, как и полагается, будет завершаться. Однако перед завершением значение STOPCODE прибавляется к SUM. Лучше образовать цикл таким образом:

```
repeat
    READ (NUMBER);
    if NUMBER <>STOPCODE then
        begin
            SUM := SUM + NUMBER;
            COUNT := COUNT + 1
        end
until NUMBER = STOPCODE;
```

Чтобы избежать появления в программах подобного рода ошибок, следует подходить к организации циклов с особой аккуратностью: следить за отсутствием ошибок как в фазе входа в цикл, так и в фазе завершения цикла. Если не делать этого, то основной причиной программных ошибок, по-видимому, станут некорректности в построении циклов.

## 7.2. Циклы со счетчиком

Циклы со счетчиком составляют такой класс, в которых выполнение исполнительной части должно повторяться заранее определенное число раз. В примере 5.3 содержится цикл со счетчиком для считывания последовательности заданной длины. Циклы со счетчиком используются довольно часто, и поэтому в языке Паскаль для этих целей имеется специальная конструкция.

Можно, конечно, циклы со счетчиком моделировать при помощи операторов *while* или *repeat*, но структура цикла со счетчиком проще. При использовании *for* ЭВМ выполняет за программиста черновую работу по инициализации управляющей переменной и по ее увеличению (уменьшению) при каждом повторении цикла. Единственное ограничение заключается в том, что тип управляющей переменной не должен быть REAL.

**Пример 7.2. Вариант неправильного употребления repeat**

```
program BADPROG;
(* Эта программа считывает ряд чисел и находит их
среднее значение. Конец последовательности чисел
должен указываться кодом окончания -1. *)
const
    STOPCODE = -1;        (* Символ конца списка *)
var
    SUM : REAL;           (* Сумма чисел *)
    NUMBER : REAL;        (* Последнее считанное чис-
                                                ло *)
    COUNT : INTEGER;      (* Размер списка *)
begin
    (* — — выдача подсказывающих сообщений — — *)
    SUM : = 0;
    COUNT : = 0;
    WRITELN ('Пожалуйста, введите ряд чисел ');
    WRITELN ('указав конец списка с помощью ',
    STOPCODE : 1);
    (* — — Считать числа — — *)
  repeat
        READ (NUMBER);
        SUM : = SUM + NUMBER;
        COUNT : = COUNT + 1
  until NUMBER = STOPCODE;
    (* — — Напечатать среднее значение — — *)
    WRITELN ('Среднее значение равно ', SUM /
    COUNT)
end.
```

Исполнительная часть цикла *for* может быть либо простым, либо составным оператором. Если начальное значение цикла *for ... to* больше конечного значения, то никакие операции не выполняются. Таким образом, следующий оператор не приведен к выполнению каких-либо операций:

```
for COUNT := 1 to 0 do
    WRITELN (COUNT)
```

Однако цикл, представленный в такой форме, распечатает целые числа от единицы до десяти:

```
for COUNT := 1 to 10 do
    WRITELN (COUNT)
```

Как можно догадаться, следующий цикл выполняет счет в обратном порядке:

```
for COUNT := 10 downto 1 do
    WRITELN (COUNT)
```

Часто исполнительная часть одного из циклов *for* является новым оператором цикла *for*. Структуры такого рода называются *вложенными циклами*. При завершении внутреннего цикла управляющая переменная внешнего цикла увеличивается. Повторение этих действий будет продолжаться до завершения внешнего цикла. Приведенный ниже вложенный цикл печатает пары чисел, начиная от (1,1), (1,2),... и кончая (10,10):

```
for LENGTH := 1 to 10 do
      for  WIDTH := 1 to 10 do
           WRITELN (LENGTH, WIDTH)
```

Управляющая переменная цикла *for* не должна изменяться какими-либо операторами внутри цикла. К ней можно обращаться и использовать ее в вычислениях, но нельзя присваивать новое значение. Присваивания могут выполняться только механизмом самого цикла. Таким образом, следующий цикл является некорректным:

```
for COUNT := 1 to 10 do
      begin
        .
        .
        .
        COUNT := COUNT - 1
      end
```

Управляющая переменная должна описываться, как и любая другая переменная. Обычно управляющая переменная имеет тип INTEGER, но в последующих главах будут рассмотрены другие типы данных, которые могут указываться в цикле *for*. Следует помнить, что управляющая переменная не может быть типа REAL.

Какое значение принимает управляющая переменная при завершении цикла? По-видимому, разумно предположить, что переменная будет равна конечному значению, определенному в операторе *for ... to*. Однако это не всегда так. Например, в приводимом фрагменте оператор WRITELN, следующий за циклом, не обязательно будет печатать значение 10.

**FOR ... TO**
**FOR ... DOWNTO**

Общая форма записи:

for *управляющая переменная :=начальное значение*
to *конечное значение*
do *оператор*
for *управляющая переменная :=начальное значение*
downto *конечное значение*
do *оператор*

При переходе к обработке оператора *for* управляющей переменной присваивается заданное начальное значение. Затем в цикле выполняется исполнительный оператор (или составной оператор). Каждый раз при выполнении исполнительного оператора управляющая переменная увеличивается на 1 (для *for...to*) или уменьшается на 1 (для *for...downto)*[1]. Цикл завершается при достижении управляющей переменной своего конечного значения.

*Примеры:*

```
for COUNT : = 1 to LISTSIZE do
    begin
        READ (NUMBER);
        SUM : = SUM + NUMBER
     end
for LENGTH : = 15 downto 1 do
     WRITELN (SQR(LENGTH))
for MULT1 : = 1 to 10 do
     for NULT2 : = 1 to 10 do
         WRITELN (MULT1, ' умноженное на ',
         MULT2, ' равно ', MULT1 * MULT2)
for RANGE : = LOWER + 1 to UPPER * 3 do
     WRITELN (SQRT (RANGE))
```

```
for COUN : = -10 to 10 do
       WRITELN (SQR (COUNT));
WRITELN (COUNT)
```

[1] Увеличение или уменьшение управляющей переменной на единицу справедливо только в том случае, если она имеет целый тип или является интервалом целого типа. В других случаях используются функции SUCC и PRED, описываемые далее. — *Прим. ред.*

Управляющая переменная COUNT будет принимать в цикле значения –10, –9, ..., +9, +10. Однако непосредственно после цикла переменная COUNT может равняться либо 10, либо 11. Такое положение объясняется своеобразием оператора *for*. Как следствие не следует использовать значение управляющей переменной за пределами цикла[1].

### 7.3. Ветвление по ряду условий

Оператор *if* позволяет программе выполнять переходы на ту или иную ветвь по значению булева условия. Используя несколько операторов *if*, можно производить ветвление по последовательности условий. В приводимом фрагменте показано, как при помощи ряда операторов *if* можно преобразовать целое число (в диапазоне 0—9) к его словесному представлению:

```
if DIGIT = 0 then
      WRITELN ('нуль');
if DIGIT = 1 then
      WRITELN ('единица');
```

Этот подход является достаточно однообразным и утомительным. Язык Паскаль предоставляет для этих целей другую управляющую структуру (оператор *case*), которая позволяет построить ветвление по ряду условий в форме, более удобной для чтения программ.

---

**CASE**

Общая форма записи:

```
    case выражение of
                  значения : оператор;
                  значения : оператор;
                        .
                        .
                        .
                  значения : оператор
    end
```

---

[1] Обычно считается, что значение управляющей переменной после «нормального» завершения цикла *for* является неопределенным. — *Прим. ред.*

Обработка оператора начинается с вычисления выражения. Выполняется оператор (или составной оператор), соответствующий результату вычисления выражения. *Примеры:*

```
case NUMBER mod 2 of
        0 : WRITELN (NUMBER, 'четное.');
        1 : WRITELN (NUMBER, 'нечетное.')
end
case MONTH of
        1, 2, 3 : WRITELN ('Первый квартал');
        4, 5, 6 : WRITELN ('Второй квартал');
        7, 8, 9 : WRITELN ('Третий квартал');
        10, 11, 12 : WRITELN ('Четвертый квартал')
end
case CODE of
        1 : for COUNT : = 1 to 5 do
                WRITELN (' ');
        2 : begin
                WRITELN (' ');
                WRITELN ('----------')
            end;
        3 :
end
```

Как и управляющая переменная в операторе цикла *for, переменная выбора (селектор)* в операторе *case* не должна принадлежать типу REAL. Селектор может быть некоторым выражением. Любому заданному значению селектора соответствует лишь один вход в списке операторов. Если селектор принимает значение, которому не соответствует ни один вход, то никакие альтернативы выполняться не будут[1]. Таким образом, если переменная ORDER равна 5, то следующий оператор *case* будет пропущен:

```
case ORDER OF
    0: WRITELN ('Чтение не выполнялось.');
    1: WRITELN ('Первое чтение.');
    2: ;
    3, 4: WRITELN ('Номер чтения', ORDER)
end
```

[1] Имеется в виду конкретная реализация. В стандарте языка Паскаль говорится, что если ни одна из констант случаев не равна текущему значению селектора, то возникает состояние ошибки и результат работы такой программы не определен. — *Прим. ред.*

Отметим, что во входе для ORDER = 2 никакой оператор не определен. Чтó произойдет, когда ORDER равно 2? Фактически вход для ORDER = 2 является пустым оператором. При ORDER, равном 2, никакие действия выполняться не будут.

---

**Пример 7.3. Преобразование цифр в слова**

```
program DIGITS;
(* Эта программа вводит целые числа в диапазоне 0—9
и печатает их в словесной форме. *)
var
    DIGIT : INTEGER; (* Ввод пользователя *)
begin
    WRITELN ('Введите, пожалуйста, одно число.');
    READ (DIGIT);
    if (DIGIT < 0) or (DIGIT > 9) then
        WRITELN ('Это число вне диапазона 0—9.')
    else
        begin
            WRITELN ('Словесное представление этого числа');
            case DIGIT of
                0 : WRITELN ('нуль');
                1 : WRITELN ('один');
                2 : WRITELN ('два');
                3 : WRITELN ('три');
                4 : WRITELN ('четыре');
                5 : WRITELN ('пять');
                6 : WRITELN ('шесть');
                7 : WRITELN ('семь');
                8 : WRITELN ('восемь');
                9 : WRITELN ('девять');
            end
        end
end.
```

---

Пример 7.3 демонстрирует одно из простых применений оператора *case*. Оператор *if* используется для гарантии того, что селектор DIGIT оператора *case* находится в пределах предполагаемого диапазона. Если пользователь случайно наберет неправильное значение, то будет напечатано короткое сообщение об ошибке.

## 7.4. Безусловные переходы

Рассмотренные управляющие структуры обеспечивают удовлетворение 99,9% потребностей программирования. Любой селективный или итеративный процесс можно запрограммировать, используя ранее определенные управляющие структуры. Тем не менее в языке Паскаль имеется средство для выполнения *безусловного перехода.*

Безусловный переход приводит к передаче управления из одного места программы в другое. Чтобы выполнить безусловный переход, необходимо сделать следующее. Во-первых, оператор, получающий управление, должен иметь *метку* в виде целого числа из диапазона от 0 до 9999 включительно. Например, оператор *end* с меткой 900 записывается следующим образом:

```
900: end
```

Далее, в начале программы следует описать эту метку аналогично образцу, приведенному ниже:

```
label 900;
```

(Описания меток должны располагаться между заголовком программы и определением констант.) И последнее, что должно быть сделано: оператор *goto* должен быть помещен в то место программы, откуда выполняется переход. Оператор *goto* передает управление на оператор с заданной меткой. Оператор *goto* можно вставлять в любое место, где могут располагаться операторы языка. Например:

```
if THETA = 360 then
        goto 900
else
        WRITELN ('Эта фигура — не окружность.')
```

Где должен использоваться безусловный переход? На этот вопрос следует ответить — «нигде». Любая программа, написанная с *goto*, может быть запрограммирована без этого оператора. Другими словами, оператор *goto* является ненужным.

Следует ли вообще использовать оператор перехода? Этот вопрос является более важным, но на него труднее ответить. В основном применение оператора безусловного перехода оправдано в двух случаях: при прежде-

временном завершении цикла *for*[1] или программы. Но даже в этих случаях можно обойтись некоторой стандартной управляющей структурой.

Не будет ничего удивительного в том, что в ваших программах почти никогда не применяется безусловный переход. В общем случае рекомендуется избегать использования этого оператора для обеспечения лучшего понимания работы программы. Бесконтрольное применение оператора *goto* может усложнить программу настолько, что разобраться в ней будет невозможно[2].

ВОПРОСЫ

*1. Сколько раз будет повторяться выполнение данного цикла?

```
for COUNT : = 1 downto 10 do
      WRITELN (COUNT)
```

2. Напишите программу, которая вводит целое число $\geqslant 2$. Программа, используя оператор цикла, должна найти сумму всех целых чисел от 1 до числа, заданного пользователем. Другими словами, программа должна вычислить значение выражения $\sum_{k=1}^{n} k$ или $1+2+$ $+3 \ldots +(n-1)+n$.

3. Предположим, что цвета голубой, красный и зеленый представляются числами 1, 2 и 3 соответственно. Напишите три фрагмента программы, которые будут печатать «Голубой», «Красный» или «Зеленый» в зависимости от значения целочисленной переменной COLOR. В первом фрагменте следует использовать вложенные операторы *if*, во втором — последовательность невложенных операторов *if*, в третьем — оператор *case*.

4. Перепишите пример 5.3, используя оператор цикла *for*.

---

[1] Не только цикла *for*, но также и циклов *WHILE* и *REPEAT*. — *Прим. ред.*

[2] В целом «проблема оператора безусловного перехода» представляется более сложной и менее однозначной, чем описано здесь. Впрочем, автор, по-видимому, и не ставил перед собой цель дать полное изложение указанной проблемы. — *Прим. ред.*

5. Сколько раз будет выполняться оператор **WRITELN** во вложенном цикле, приведенном ниже?

```
for COUNT1 : = 1 to 4 do
        for COUNT2 : = 1 to 8 do
              WRITELN (COUNT1 * COUNT2)
```

# Глава 8
# Типы данных, определяемые программистом

## 8.1. Перечисляемый тип

Ранее в книге были описаны три типа данных: REAL, INTEGER и BOOLEAN. Используя эти типы, можно работать с числами с плавающей точкой (например, 3,1415 или 2,718), с целыми числами (например, 152 или -3) и булевыми значениями TRUE и FALSE. Однако иногда приходится работать с данными, которые не попадают ни в одну из этих категорий.

Допустим, необходимо представить последовательность месяцев года. Если мы ограничены встроенными типами языка Паскаль, то можно использовать прием, который ставит в соответствие число 0 — январю, 1 — февралю, 2 — марту и т. д. Для того чтобы присвоить переменной THISMONTH значение «июнь», можно написать

```
THISMONTH := 5
```

Очевидно, это решение несколько неудобно. Оно заставляет нас делать умственные усилия, чтобы трактовать числовые значения. На самом деле удобнее было бы написать

```
THISMONTH := JUNE
```

Другими словами, хочется иметь средства, позволяющие создавать дополнительные типы данных. Язык Паскаль позволяет это сделать. Приведем описание двух переменных — THISMONTH и LASTMONTH — значениями которых являются месяцы года:

```
THISMONTH, LASTMONTH : (JAN, FEB, MARCH,
                        APRIL, MAY, JUNE,
                        JULY, AUG, SEPT, OCT,
                        NOV, DEC)
```

В этом случае создается новый тип данных, состоящий из значений, заключенных в скобки. Это пример данных *перечисляемого типа.* Перечисляемый тип — это определен-

ный программистом тип, составленный из множества упорядоченных[1] элементов. Приведем еще несколько описаний, которые создают перечисляемый тип.

```
SUNSIGN:  (AQUARIUS, PISCES, ARIES, TAURUS,
          CEMINI, GANÇER, LEO, VIRGO, LIBRA,
          SCORPIO, SAGITTARIUS, CAPRICORN)
TODAY:  (SUNDAY, MONDAY, TUESDAY, WEDNES-
        DAY, THURSDAY, FRIDAY, SATURDAY)
SEX: (MALE, FEMALE)
```

Может показаться странным, что переменная представляет месяц, тем не менее THISMONTH и LASTMONTH осуществляют именно это. Переменной перечисляемого типа может быть назначено любое «значение» в пределах типа. В программе, где SUNSIGN описана так, как показано выше, можно написать следующий оператор присваивания:

```
SUNSIGN := PISCES
```

Переменные перечисляемых типов могут быть, как и переменные других типов, использованы в булевых выражениях:

```
if THISMONTH = OCT then
    WRITELN ('Октябрь.')
```

Могут быть использованы также операции сравнения

```
if THISMONTH > SEPT then
    WRITELN ('Четвертый квартал.')
```

Здесь может возникнуть интересный вопрос: как один элемент перечисляемого типа может быть больше или меньше другого? Другими словами, каким образом FEB может быть больше или меньше, чем SEPT? Ответ на этот вопрос дает факт упорядоченности элементов в описании. Упорядоченность типа данных автоматически устанавливает отношение порядка между элементами. Самый левый в описании элемент имеет минимальное

---

[1] Упорядоченность элементов определяется порядком из следования. Наличие этого свойства позволяет ввести функции получения следующего или предыдущего элемента в рамках данного перечисляемого типа. — *Прим. ред.*

значение, а наиболее правый — максимальное. Для переменной TODAY это означает, что

SUNDAY < MONDAY < ... < FRIDAY < SATURDAY

Переменные перечисляемого типа можно использовать в любых управляющих структурах языка Паскаля. Вот, например, оператор *for*, который «пересчитывает» от SEPT до DEC:

```
for LASTMONTH : = SEPT to DEC do
    begin
      .
      .
      .
    end
```

В пределах цикла LASTMONTH принимает последовательно значение SEPT, OCT, NOV и DEC. А вот оператор *case*, чьим селектором является переменная перечисляемого типа:

```
case THISMONTH of
    JAN, FEB, MARCH : WRITELN ('Первый квартал.');
    APRIL, MAY, JUNE : WRITELN ('Второй квартал.');
    JULY, AUG, SEPT : WRITELN ('Третий квартал.');
    OCT, NOV, DEC : WRITELN ('Четвертый квартал.')
end
```

При составлении имен переменных перечисляемого типа следует руководствоваться правилами, которые применяются к переменным и символическим константам. Они не могут, например, начинаться с цифры и состоять из чего-либо, кроме букв, цифр или символов подчеркивания. Они не могут быть зарезервированными словами. Так, следующее описание неверно:

```
DUMMY : (3STAR, HELLO?, BEGIN)
```

Кроме того, важно запомнить, что все идентификаторы программы на языке Паскаль должны быть уникальными. Переменные не могут иметь тех же имен, что и символические константы, так как иначе не будет возможности обращаться к ним. Подобно этому, имена элементов перечисляемого типа не должны вступать в конфликт с другими идентификаторами программы. По

этой причине нижеследующие описания, если они появятся в одной программе, будут отвергнуты:

```
BLUE : INTEGER;
COLOR : (RED, GREEN, BLUE);
```

### 8.2. Интервальный тип

Предположим, нам нужно обрабатывать информацию о том, сколько дней человек проработал в определенной неделе. С этой целью можно описать переменную DAYS _ WORKED типа INTEGER. Если человек вообще не приходил на работу, DAYS _ WORKED нужно установить в нуль. В другом крайнем случае, когда человек работал каждый день, DAYS _ WORKED нужно сделать равным семи.

Для ситуаций, подобных этой, в языке Паскаль имеется *интервальный тип.* Интервальный тип является отрезком какого-то другого (называемого *базовым*) типа. Примером описания, которое создает переменную интервального типа, является такая запись:

```
DAYS _ WORKED : 0..7
```

Эта запись определяет нижнюю и верхнюю границы интервала. В приведенном описании базовым является тип INTEGER. Теперь DAYS _ WORKED может быть использована как обычная целочисленная переменная с одним лишь исключением: ей не может присваиваться значение вне диапазона 0..7, т. е. она не может быть меньше 0 или больше 7. Если ей будет присваиваться значение, расположенное вне диапазона, в результате будет выработано условие ошибки.

В качестве базового типа нельзя использовать тип REAL. Однако в качестве базового может быть использован перечисляемый тип. Предполагая, что переменная TODAY описана, как в разд. 8.1, видим, что описание

```
WEEKDAY : MONDAY .. FRIDAY
```

создает интервал, у которого базовый тип — это (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY).

Верхняя и нижняя границы интервала могут быть определены либо с помощью литеральной константы

(например, –1 или 16), либо символической константы (например, LISTSIZE или STOPCODE). Для того чтобы проиллюстрировать применение символических констант, приведем последовательность описаний некоторой гипотетической программы:

```
const
    MINVALUE = -10; (* нижняя граница *)
    MAXVALUE = 25; (* верхняя граница *)
var
    NUMBER : MINVALUE .. MAXVALUE;
```

Теперь NUMBER — целочисленная переменная, которая может принимать значения из диапазона –10...25. Использование символических констант для определения интервала — это обычное явление, в особенности когда границы диапазона неизвестны или неочевидны.

В программе примера 8.1 переменная YEAR описана как интервал с нижней границей 1582 и верхней границей 9999. Если пользователь введет число вне этого интервала, то будет напечатано сообщение об ошибке и работа программы будет прервана. Если пользователь введет правильное число, программа исследует его с целью определить, представляет ли оно високосный год.

---

**Пример 8.1. Программа определяет, является ли год високосным**

```
program LEAPYEAR;
(* В этой программе вводится год по грегорианскому
календарю. После этого определяется, является ли год
високосным. *)
var
    YEAR : 1582..9999;
    EXTRADAY : BOOLEAN;
begin
    (* — — Ввод года — — *)
    WRITELN ('Какой год вы хотите проверить?');
    READ (YEAR);
    (* — — Високосный ли этот год? — — *)
    EXTRADAY : = FALSE;
    if (YEAR mod 4 = 0) and (YEAR mod 100<>0) then
        EXTRADAY : = TRUE;
    if YEAR mod 400 = 0 then
        EXTRADAY : = TRUE;
```

```
    (* — — Печать результата — — *)
    if EXTRADAY then
        WRITELN (YEAR : 4, ' — високосный год.')
    else
        WRITELN (YEAR : 4, ' — не високосный год.')
end.
```

---

Интервальный тип полезен по двум причинам. Во-первых, он повышает наглядность программы. Когда мы определяем, какие значения переменная может иметь, мы задаем полезную вспомогательную информацию о назначении переменной. Во-вторых, сообщения об ошибках, сигнализирующих о выходе из интервала, помогают найти необнаруженные ранее ошибки в логике программы[1].

## 8.3. Описания типов

В языке Паскаль есть два способа описания типов. Первый способ состоит в том, что тип определяется при описании переменных. Этот метод использовался до сих пор. Например, чтобы определить переменную COLOR перечисляемого типа, можно задать следующее описание;

```
COLOR : (RED, GREEN, BLUE)
```

Существует также и второй способ определения типа данных. Если необходимо, можно описать тип данных отдельно и определить его имя. Вот пример описания, которое создает тип данных с именем PALETTE:

```
PALETTE = (RED, GREEN, BLUE)
```

Теперь PALETTE такой же идентификатор типа, как и INTEGER или BOOLEAN. Описание типа находится между определением констант и описанием переменных. Предполагая, что тип PALETTE описан, как это было показано выше, можно задать такое описание переменной:

```
COLOR : PALETTE
```

---

[1] С точки зрения реализации указание интервала полезно также и тем, что позволяет более экономно распределять память для хранения переменных интервального типа. — *Прим. ред.*

Теперь COLOR имеет тип (RED, GREEN, BLUE). Заметим, что описание типа использует знак равенства, а не двоеточие. Описанию типов в программе предшествует зарезервированное слово *type*.

Программа, показанная в примере 8.2, является модификацией примера 8.1. В ней описан тип GREGORIAN, который является интервалом типа INTEGER. После этого описывается переменная YEAR, имеющая тип GREGORIAN.

---

**Пример 8.2. Описание типов данных**

```
program LEAP2;
(* В этой программе вводится год по грегорианскому
календарю. После этого определяется, является ли год
високосным. *)
type
    GREGORIAN = 1582..9999;
var
    YEAR : GREGORIAN;
    EXTRADAY : BOOLEAN;
begin
    (* — — Ввести год — — *)
    WRITELN ('Какой год вы хотите проверить?');
    READ (YEAR);
    (* — — Это високосный год? — — *)
    EXTRADAY : = FALSE;
    if (YEAR mod 4 = 0) and (YEAR mod 100 <> 0) then
        EXTRADAY : = TRUE;
    if YEAR mod 400 = 0 then
        EXTRADAY : = TRUE;
    (* — — Напечатать результат — — *)
    if EXTRADAY then
        WRITELN (YEAR : 4, ' — это високосный год.')
    else
        WRITELN (YEAR : 4,' — это не високосный
        год.')
end.
```

---

## 8.4. Свойства порядковых типов

Все типы данных, которые обсуждались до сих пор, за одним исключением, имеют общее свойство: они могут быть представлены как перечисляемый тип. Вот,

например, как могли бы быть определены BOOLEAN и INTEGER:

```
BOOLEAN = (TRUE, FALSE);
INTEGER = (-MAXINT .. .MAXINT);
```

Типы данных, которые могут быть определены таким образом, называются *порядковыми типами.* Типы BOOLEAN и INTEGER, так же как и перечисляемые типы, относятся к порядковым. Однако тип REAL не является порядковым.

Разница между порядковыми и непорядковыми типами очень существенна. Переменная или выражение, которые используются в качестве индекса в цикле *for*, должны быть порядкового типа; то же утверждение справедливо и для селектора оператора *case.* Базовый тип интервала также должен быть порядковым.

За исключением минимального и максимального элементов, каждый элемент порядкового типа имеет предыдущий (*предшественник*) и последующий (*преемник*) элементы. У целой величины 4 имеется предшественник 3 и преемник 5. В перечисляемом типе (JAN, FEB, MARCH, ..., NOV, DEC) у FEB предшественником является JAN, а преемником — MARCH. У минимального элемента JAN нет предшественника, а у максимального элемента DEC нет преемника. Для того чтобы обеспечить более гибкое манипулирование значениями порядкового типа, в язык Паскаль включены функции PRED и SUCC. Аргументами этих функций могут быть выражения порядкового типа, а результатом выполнения является соответственно его предшественник либо преемник. Будет ошибкой использовать такую запись:

```
THISMONTH := JAN + 1
```

Можно найти элемент, следующий за JAN, написав:

```
THISMONTH := SUCC(JAN)
```

Попытка вычислить неопределенные выражения вроде PRED(JAN) или SUCC(DEC) приведет к возникновению ошибки. Однако перечисляемый тип можно обходить по кольцу, если использовать оператор *if*:

```
if THISMONTH = DEC then
    NEXTMONTH : = JAN
else
    NEXTMONTH : = SUCC (THISMONTH)
```

Каждый элемент порядкового типа имеет свой собственный *порядковый номер*. Для значений типа INTEGER (или его интервала) порядковый номер является самим значением. Для упорядоченного типа, содержащего $n$ элементов, каждый элемент имеет порядковый номер между 0 и $n-1$. Первый элемент имеет порядковый номер 0, последний имеет порядковый номер $n-1$. В язык Паскаль включена специальная функция ORD, значением которой является порядковый номер выражения порядкового типа. Таким образом,

```
ORD(100) = 100
ORD(PRED(100)) = 99
ORD(JAN) = 0
ORD(DEC) = 11
ORD(MARCH) = 2
ORD(SUCC(FEB)) = 2
```

**ВОПРОСЫ**

*1. Предполагая, что сделаны описания, подобные тем, которые приведены в разд. 8.1, определите, какие из нижеследующих операторов присваивания неверны (COUNT имеет тип INTEGER):

```
THISMONTH : = MARCH
SUNSIGN : = FEMALE
TODAY : = ORD(SEX)
SUNSIGN : = SUCC(SUNSIGN)
COUNT : = ORD(ORD(TODAY))
```

*2. При тех же предположениях относительно описаний из разд. 8.1 определите, какие из следующих дополнительных определений содержат ошибки:

```
WORKINGDAY : FRIDAY..MONDAY
WHOLE : 0..MAXINT
NEXTMONTH : JAN..DEC
```

3. Напишите программу, которая вводит пять групп семестровых курсовых оценок (по 4-балльной шкале)

с количеством часов, выделенных для каждого курса. Программа печатает суммарную оценку успеваемости и среднюю оценку по каждому семестру.

4. Вначале опишите переменную SEASON, возможными значениями которой являются названия четырех времен года. Затем напишите программу, которая, введя два целых числа, представляющих день и месяц, присваивает переменной SEASON правильное значение, соответствующее введенным данным.

# Глава 9
# Массивы

## 9.1. Одномерные массивы

Рассматриваемые до сих пор типы данных являлись *простыми* типами. Простой — это такой тип, который может представлять только одно значение. Например, переменная типа INTEGER может хранить только одно целое число.

Язык Паскаль содержит наряду с этим и *структурированные* типы, которые могут представлять совокупности значений. Наиболее часто используемым структурированным типом является *массив.* Так выглядит описание переменной, представляющей массив:

```
LIST : array [1..10] of INTEGER
```

LIST — массив, содержащий 10 значений типа INTEGER. К первому элементу массива можно обратиться, указав LIST[1], к следующему — LIST[2] и т. д. Пример цикла, который присваивает всем компонентам LIST значение 0, представляется следующим образом:

```
for COUNT : = 1 to 10 do
    LIST[COUNT] := 0
```

Теперь компоненты массива LIST можно изобразить так:

```
LIST[1]   0
LIST[2]   0
LIST[3]   0
  .       .
  .       .
  .       .
LIST[10]  0
```

Значения внутри квадратных скобок называются *индексами;* они используются для указания конкретной компоненты массива. Индекс может быть произвольным

выражением. В приведенном примере и индекс, и компонента принадлежит типу INTEGER.

Однако в зависимости от способа, которым был описан массив, его компоненты и индекс могут быть различных типов.

---

**Описание одномерных массивов**

Общая форма записи:
*Переменная:* array [*тип индекса*] of *тип компонент.*
Тип индекса должен быть порядковым.
Тип компонент может быть любого типа.
*Примеры:*
GRADES : array [1.. CLASSSIZE] of REAL
WEIGHTS : array [0..15] of INTEGER
MAXTEMPS : array [JAN..DEC] of –20...110

---

*Тип компонент* массива — это просто тип данных, ассоциированный с каждой компонентой массива. Например, если массив имеет тип компонент INTEGER, то каждая его компонента может быть использована как целочисленная переменная.

*Тип индекса* определяет границы изменения индекса. Если сделана попытка использовать несуществующую компоненту, то возникает ошибка. Ошибки такого сорта называются ошибками *неверного индекса.* Если LIST был описан, как показано выше, каждый из нижеследующих операторов приведет к ошибке неверного индекса:

```
LIST [–2] := 0
WRITELN(LIST [3.14])
LIST [11] := LIST [11] + 2
```

Если два массива имеют одинаковые типы индексов и одинаковые типы компонент, можно проверить их равенство или неравенство с помощью одного булева сравнения. Однако в операциях с массивами могут быть использованы только знаки равно (=) и не равно (< >). Другие операции отношения, такие, как больше чем (>) или меньше чем (<), должны использоваться при поэлементном сравнении с помощью какого-либо вида цикла.

Если читателя интересует, как правильно читать обозначение индекса, заметим, что индекс добавляется к идентификатору массива как следующее за ним опре-

**Пример 9.1. Табулирование результатов выборов**

```
program BALLOTS;
(* Программа читает последовательности бюллетеней,
каждый из которых представлен с помощью целого
значения. Результаты голосования распечатываются. *)
const
    STOPCODE = -1; (* Обозначает конец ввода *)
    CANDIDATES = 3; (* Число кандидатов *)
var
    VOTES : array [1..CANDIDATES] of INTEGER;
    COUNT : INTEGER;
    ONEVOTE : INTEGER;
begin
    (* -- Каждый начинает с нулевым числом голо-
    сов -- *)
    for COUNT : = 1 to CANDIDATES do
            VOTES [COUNT]: = 0;
    (* -- Чтение бюллетеня -- *)
    WRITELN ('Каждый бюллетень должен быть пред-
    ставлен');
    WRITELN ('числом между 1 и ', CANDIDATES : 1,
    ', ');
    WRITELN ('Введите, пожалуйста, бюллетени, за-
    вершая');
    WRITELN ('список значением ', STOPCODE : 2, '.');
    READ (ONEVOTE);
    while ONEVOTE <> STOPCODE do
          begin
             if (ONEVOTE < 1) or (ONEVOTE > CANDI-
             DATES) then
             WRITELN ('Бюллетень, закодированный',
             ONEVOTE, 'не учитывается.')
             else
                 VOTES[ONEVOTE] : = VOTES
             [ONEVOTE] + 1; READ(ONEVOTE)
          end;
    (* -- Распечатка результатов -- *)
    WRITELN (' ');
    WRITELN ('Результаты голосования следующие: ');
    WRITELN (' ');
    for COUNT : = 1 to CANDIDATES do
            WRITELN ('Кандидат № ', COUNT : 2,
              'получил', VOTES[COUNT], 'голосов.')
end.
```

деление. Таким образом, переменная с индексом LIST [7] произносится как «LIST седьмой».

Программа, приведенная в примере 9.1, использует массив для накопления распределения голосов за кандидатов на выборах. Каждая компонента VOTES содержит общее число голосов, полученных определенным кандидатом. VOTES [1] содержит число голосов за кандидата № 1, VOTES[2] — число голосов за кандидата № 2 и т. д. Если программа читает значение за пределами ожидаемых границ, печатается сообщение, и голос отбрасывается.

Так как в качестве типа индекса массива часто используется интервальный тип (например, 1..10 или JAN..DEC), можно специфицировать границы индекса с помощью символических констант. В этом примере идентификатор CANDIDATES — символическая константа, равная 3. Для того чтобы изменить число кандидатов в списке, нужно изменить только определение константы (однако невозможно изменить размер массива во время выполнения программы).

## 9.2. Некоторые типичные операции над массивами

Допустим, у нас имеется массив с именем ITEMS, который описан следующим образом:

```
ITEMS : array [0..63] of REAL
```

Какие операции над ним можно выполнять? Мы уже рассматривали, каким образом можно распечатать компоненты массива или присвоить им значения. Однако это лишь часть возможных операций над массивами. Рассмотрим, какие еще действия над массивами возможны.

- *Поиск.* Если, например, массив LIST описан, как указано выше, можно узнать, сколько отрицательных чисел содержится в массиве или содержится ли в массиве значение 2.0? Для того чтобы ответить на вопросы, подобные этим, программа должна производить поиск в массиве. В приводимом ниже фрагменте программы отыскивается наибольшее значение в ITEMS:

```
MAXVALUE : = ITEMS[0];
for COUNT : = 0 to 63 do
        if ITEMS[COUNT] > MAXVALUE then
                MAXVALUE : = ITEMS[COUNT];
```

Вначале MAXVALUE присваивается значение ITEMS [0]. Затем в цикле *for* каждая компонента ITEMS исследуется, чтобы определить, не содержит ли она большее число. Если найдена компонента, большая чем MAXVALUE, значение MAXVALUE заменяется на найденное. Таким образом, в конце цикла MAXVALUE будет равно наибольшей величине в массиве.

• *Копирование.* Иногда необходимо скопировать один массив в другой. Если массив BACKUP описан так же, как был описан ITEMS, копирование всех компонент ITEMS в соответствующие позиции массива BACKUP можно произвести следующим образом:

```
for COUNT := 0 to 63 do
    BACKUP[COUNT] := ITEMS[COUNT]
```

Теперь два массива содержат идентичные компоненты. Для того, чтобы упростить эту операцию, в языке Паскаль допускается копирование содержимого идентично описанных массивов с помощью одного оператора присваивания. Например:

```
BACKUP := ITEMS
```

Предыдущее содержимое BACKUP будет уничтожено и заменено содержимым ITEMS. Сам массив ITEMS при этом не изменяется.

• *Численные оценки.* Допустим, в массиве ITEMS представлены значения суммарной еженедельной продажи (в долл.) 64 видов продукции некоторой фирмы. Имея массив с соответствующими числами, вы можете с его помощью произвести определенные численные оценки. Одной из возможных оценок может быть сумма компонент, содержащихся в массиве еженедельной продажи продуктов производства:

```
WEEKLYVOLUME : = 0
for COUNT : = 0 to 63 do
        WEEKLYVOLUME : = WEEKLYVOLUME +
        + ITEMS[COUNT];
```

Другие оценки могут состоять в вычислении некоторых статистических величин, например математического ожидания или стандартного отклонения (дисперсии). Ниже приведен фрагмент программы, которая вычисляет математическое ожидание (т. е. среднее значение) компонент ITEMS:

```
SUM : = 0
for COUNT : = 0 to 63 do
      SUM : = SUM + ITEMS [COUNT];
MEAN : = SUM / 64;
```

• *Перестановки компонент.* В некоторых случаях необходимо поменять два элемента массива местами. Поменять местами их значения, например присвоить ITEMS[3] значение ITEMS[4] и наоборот. На первый взгляд может показаться, что последовательность операторов, приведенная ниже, осуществляет перестановку компонент:

```
ITEMS[3] := ITEMS[4];
ITEMS[4] := ITEMS[3];
```

Однако она этого не делает. После выполнения этих операторов ITEMS[3] и ITEMS[4] будут иметь одинаковые значения, равные первоначальному значению ITEMS[4]. Ниже приводится последовательность операторов, которая решает эту задачу:

```
TEMP := ITEMS[3]
ITEMS[3] := ITEMS[4]
ITEMS[4] := TEMP
```

Переменная TEMP выступает в роли промежуточной памяти (предполагается, что TEMP была описана в соответствии с типом компонент ITEMS). В этом примере TEMP используется, чтобы сохранить значение ITEMS[3] в процессе перестановки.

## 9.3. Многомерные массивы

Массив ITEMS из разд. 9.2 — одномерный массив. Это означает, что для указания определенной компоненты достаточно указать один индекс.

Однако мы не ограничены одномерными массивами. При описании можно указать любую размерность массива. Приведенная ниже последовательность описаний создает двумерный массив.

```
const
    ROWS = 5;
    COLUMNS = 7;
var
    TABLE: array [1..ROWS, 1..COLUMNS] of INTEGER;
```

Двумерный массив (например, TABLE) аналогичен математическому понятию *матрицы*. TABLE содержит 35 компонент, каждая из которых поставлена в соответствие единственной паре индексов. Ниже приводится вложенный цикл, который устанавливает все компоненты TABLE в нуль.

```
for HEIGHT : = 1 to ROWS do
        for WIDTH : = 1 to COLUMNS do
                TABLE [HEIGHT, WIDTH] : = 0
```

Каждый раз, когда внутренний цикл завершается, внешний цикл увеличивает HEIGHT на единицу, и внутренний цикл выполняется вновь. Таким образом, все компоненты массива TABLE, начиная с TABLE[1,1], TABLE[1,2], TABLE[1,3] и т. д. до TABLE[5,7], становятся равными нулю. Теперь содержимое TABLE можно изобразить следующим образом:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Типы обоих индексов TABLE являются интервалами типа INTEGER. Но, как и в случае одномерных массивов, типы индексов принадлежат к порядковым типам данных. Например, следующая последовательность операторов описания является корректной:

```
type
    SUITE = (CLUBS, HEARTS, DIAMONDS, SPADES);
    FACEVALUE = 2..14;
    DECK = array [SUITE, FACEVALUE] of BOOLEAN;
var
    DEALT : DECK;
```

Теперь DEALT является переменной типа DECK. Она содержит 52 компоненты — по одной на каждую карту колоды. Поскольку каждая компонента имеет тип BOOLEAN, то данный массив позволяет проследить, использовалась или нет та или иная карта. Чтобы

определить состояние восьмерки червей, можно записать так: DEALT[HEARTS, 8]. Ниже приводится фрагмент программы, которая распечатывает состояние каждой карты:

```
for SUITECOUNT : = CLUBS to SPADES do
      for FACECOUNT : = 2 to 14 do
      begin
            case SUITECOUNT of
              CLUBS : WRITE(FACECOUNT : 2,
              'треф');
              HEARTS : WRITE(FACECOUNT : 2,
              'червей');
              DIAMONDS : WRITE(FACECOUNT : 2,
              'бубен');
              SPADES : WRITE(FACECOUNT : 2,
              'пик')
            end;
            if DEALT[SUITECOUNT, FACECOUNT]
            then
              WRITELN ('используется.')
            else
              WRITELN ('не используется.')
      end;
```

Несмотря на то что трудно представить себе массивы более двух или трех измерений, конструировать массивы можно любой размерности. Для ЭВМ безразлично, что обрабатывать: пятимерный или двумерный массив. Так что, как и в случае одно- и двумерных массивов, тип индексов для пятимерного массива должен быть порядковым.

## 9.4. Упакованные массивы

Существует несколько форматов для представления массивов в памяти ЭВМ. Некоторые из них позволяют запоминать массивы в меньшем объеме памяти, однако ценой более медленного выполнения программы. Пользователь по своему усмотрению может использовать тот или иной формат. Массив, запоминаемый в таком «компактном» формате, называется *упакованным* массивом. Вот описание, создающее упакованный массив:

```
PLAYERS : packed array [1 .. 10] of SCORES
```

Любая операция, которая выполняется над неупакованным массивом, может быть выполнена и над упакованным массивом, за исключением двух. Во-первых, нельзя производить булево сравнение двух упакованных массивов, чтобы определить их равенство или неравенство. Во-вторых, компоненты упакованного массива не могут быть введены пользователем с помощью операторов READ или READLN. Кроме того, операции над упакованными массивами выполняются медленнее, чем над неупакованными. Эта разница в скорости объясняется тем, что доступ к компоненте неупакованного массива производится быстрее, чем к компоненте упакованного массива.

В каком случае предпочтительнее использовать упакованный массив вместо неупакованного? Большинство программистов используют неупакованные массивы как само собой разумеющееся до тех пор, пока не обнаруживают, что программа занимает слишком большой объем памяти. Имеются все основания выполнять упаковку больших, редко используемых массивов. Упаковка маленьких массивов дает малую экономию памяти; упаковка часто используемых массивов может в значительной степени замедлить программу.

ВОПРОСЫ

*1. Приведите описание типа, которое определяет одномерный массив:

а) с индексами типа BOOLEAN и компонентами типа REAL;
б) с индексами типа HUES и компонентами типа INTEGER;
в) с индексами типа 1 .. 10 и типом компонент 1 .. 10;

*2. Какие из следующих описаний переменных неверны?

```
HEIGHTS : array [1 .. PERSONS] of 5.0 .. 6.9
STATUS : array [10 .. 1] of BOOLEAN
VECTOR : array [1 .. 2] of REAL
SALES : array [10 .. 10, 3 .. 3] of INTEGER
```

3. Используя массив ITEMS разд. 9.2, напишите фрагмент программы, которая изменяет на обратный порядок следования компонент ITEMS, т. е. нужно поменять

местами первую и последнюю компоненты ITEMS, вторую и предпоследнюю и т. д.

4. Напишите программу, которая вводит значения ежемесячной зарплаты для заданного двенадцатимесячного периода и затем определяет месяц, в котором:

а) обнаружен наибольший прирост зарплаты;
б) обнаружено наибольшее уменьшение зарплаты.

5. Напишите программу, которая вводит два массива MATRIX1 и MATRIX2, описанные следующим образом:

```
const
    N = 4
var
    MATRIX1, MATRIX2 : array[1..N, 1..N] of INTEGER;
```

Затем программа вычисляет и печатает произведение матриц. Произведение двух N×N матриц представляет собой массив такого же типа. Компоненты произведения могут быть определены следующим образом:

```
PRODUCT [I, J] = MATERIX1 [I, 1] * MATRIX2 [1, J]
               + MATRIX1 [I, 2] * MATRIX2 [2, J]
               + MATRIX1 [I, 3] * MATRIX2 [2, J]
                     .
                     .
                     .
               + MATRIX1 [I, N] * MATRIX2 [N, J]
```

# Глава 10.
# Типы данных, представляющие слова

## 10.1. Тип данных CHAR

В большинстве применений ЭВМ алфавитно-цифровая информация используется наряду с числовой информацией. Например, программа для ведения списков почтовых отправлений должна иметь возможность воспринимать алфавитно-цифровые знаки в качестве входных данных, запоминать и распечатывать их. Прежде чем мы сможем написать программу, которая манипулирует алфавитно-цифровыми знаками (литерами), нам потребуется тип данных для их представления. Для этих целей в языке Паскаль предусмотрен тип данных CHAR.

Так же, как переменная типа INTEGER может хранить одно целое число, переменная типа CHAR может хранить одну литеру. Если переменная ALPHA описана как переменная типа CHAR, то следующие операторы присваивания верны:

```
ALPHA := 'P'
ALPHA := '+'
ALPHA := '3'
ALPHA := ' '
ALPHA := ''''
```

Первый оператор присваивания делает ALPHA равной литере *P*. Второй делает ALPHA равной литере плюса (+). Третий делает ALPHA равной литере 3. (Заметим, что литера *3* отличается от целого 3. Когда цифра *3* заключена в апострофы, то *3* — такая же литера, как и любая другая. Таким образом, ALPHA не может быть использована в арифметических действиях, даже если ее значение случайно изображает цифру.)

Четвертый оператор присваивания делает ALPHA равной литере пробела. Хотя литера пробела при печати не изображается, она является нормальным значением типа CHAR. Последний оператор присваивания делает

ALPHA равной литере апострофа; это специальный случай, так как знак апострофа используется для ограничения значения типа CHAR.

Набор значений типа CHAR, доступных на данной ЭВМ, называется *множеством литер* этой ЭВМ. Множество литер обычно включает в себя заглавные и строчные буквы, цифры от 0 до 9 и ряд других символов. ЭВМ, на которых устанавливается компилятор с языка Паскаль, обычно используют код, называемый ASCII (американский стандартный код обмена информацией[1]). Отношения упорядочения значений типа CHAR определяется множеством литер; упорядочение литер ASCII представлено в приложении В. Важно, что соблюдаются следующие отношения:

'A' < 'B' < 'C' < ... < 'X' < 'Y' < 'Z'
'0' < '1' < '2' < ... < '7' < '8' < '9'

Для проверки равенства или неравенства переменных типа CHAR могут использоваться операторы булевого сравнения. В примере 10.1 приведена программа, которая считывает две литеры и печатает больше, равна или меньше первая литера второй.

---

**Пример 10.1. Сравнение двух литер**

```
program COMPARE;
(* Программа читает две литеры. Затем она определяет,
больше, равна или меньше первая литера второй.*)
var
      FIRST, SECOND : CHAR; (*входные литеры *)
begin
      READ (FIRST, SECOND);
      WRITELN ('Первая литера');
      if FIRST > SECOND then
            WRITELN ('больше второй.')
      else
            if FIRST = SECOND then
                  WRITELN ('равна второй.')
            else
                  WRITELN ('меньше второй.')
end.
```

---

[1] Помимо указанного кода широко используется также код EBCDIC (расширенный двоично-десятичный информационный код), а в СССР — системы ГОСТ и ISO. — *Прим. ред.*

Так как CHAR — порядковый тип, функции, описанные в разд. 8.4, применимы к значениям типа CHAR. Например, функции SUCC и PRED возвращают следующий и предшествующий символы литерного множества. SUCC ('0') рвен '1', потому что '1' следует за '0'.

Порядковый номер значения типа CHAR называется также *кодом литеры*. В ASCII код литеры 'A' равен 65; код литеры '3' равен 51 (другие коды литер ASCII приведены в приложении В). Функция ORD возвращает значение кода литеры для любого значения типа CHAR. Например, если переменная CHAR типа ALPHA равна 'A', то

ORD(ALPHA) = ORD('A') = 65.

В языке Паскаль имеется функция CHR, которая получает целочисленный аргумент и возвращает соответствующую литеру. На ЭВМ, использующей код ASCII, CHR (65) равен 'A'. Так так функция CHR является обратной по отношению ORD, то CHR(ORD(ALPHA)) = = ALPHA. Если вызвать CHR с целым аргументом, который не является правильным кодом литеры, то это вызовет ошибку.

В языке Паскаль отсутствуют функции преобразования типов, которые преобразуют величины типа INTEGER непосредственно в значении типа CHAR и наоборот. Однако существуют возможности подобных преобразований. Если COUNT является целым значением от 0 до 9 включительно, можно преобразовать его в значение типа CHAR с помощью следующего оператора присваивания:

```
ALPHA := CHR(COUNT + ORD('0'))
```

И обратно, если ALPHA содержит цифровую литеру, можно преобразовать ее в значение типа INTEGER, записав

```
COUNT := ORD(ALPHA) - ORD('0')
```

Тот факт, что CHAR является порядковым типом, означает, что переменная типа CHAR может быть управляющей переменной для цикла *for* или селектором в операторе *case*. В приведенном ниже цикле *for* печатаются все литеры от 'A' до 'Z':

```
for ALPHA : = 'A' to 'Z' do
        WRITE(ALPHA)
```

Программа, приведенная в примере 10.2, использует массив, тип индекса которого является интервалом типа CHAR. Эта программа определяет частоту, с которой буквы алфавита появляются в тексте. После считывания текста каждая компонента массива содержит число появлений соответствующей литеры, причем в качестве индекса элемента используется сама литера.

---

**Пример 10.2. Распределение частот появления букв**

```
program LETTERFREQ;
(*Эта программа считывает текст и определяет частоту
появления для каждой буквы.*)
var
    LETTERS : array ['A'..'Z'] of INTEGER;   (*частоты*)
    SYMBOL : CHAR;              (*входная литера*)
    INDEX : CHAR;               (*индекс цикла*)
begin
(*--Чтение текста--*)
    while not EOF(INPUT) do
        begin
            READ(SYMBOL);
            if (SYMBOL >= 'A') and (SYMBOL <= 'Z')
            then
            LETTERS[SYMBOL] := LETTERS[SYMBOL]
            + 1
        end;
(*--Печать результата--*)
    WRITELN('Литера  Частота');
    for INDEX := 'A' to 'Z' do
        WRITELN(INDEX,  LETTERS[INDEX] : 15))
end.
```

---

Подобно другим типам данных, CHAR может быть использован в определениях констант. Ниже приведен пример определения символической константы типа CHAR:

```
const
    BLANK = ' ';
    PLUS  = '+';
    MINUS = '-';
```

Идентификаторы BLANK, PLUS и MINUS теперь определены так, как если бы это были определяющие их литеры. Определять константы типа CHAR в особен-

ности полезно, если некоторая литера выбрана в качестве указателя специфического условия. Например, в программе LETTERFREQ литера '@' может быть использована для указания конца вводимого текста. Для наглядности литере «конец текста» можно присвоить смысловое имя. Таким образом,

```
const
    FINISHED = '@';
    .
    .
    .
    READ(SYMBOL);
    while SYMBOL <> FINISHED do
        begin
            if (SYMBOL >= 'A') and (SYMBOL <=
            <= 'Z') then
          LETTERS[SYMBOL] := LETTERS[SYMBOL]
          + 1
            READ(SYMBOL)
        end;
```

### 10.2. Массивы литер

*Строка* — это последовательность литер. Литерные строки уже использовались в качестве аргументов операторов WRITE и WRITELN. Приведенный ниже оператор WRITELN печатает на терминале пользователя строку

```
WRITELN ('Быть или не быть?')
```

Эта строка содержит 17 литер. С точки зрения программиста, можно представить эту строку в качестве массива литер, другими словами, как массив, компоненты которого имеют тип CHAR. Переменную данного типа можно описать таким образом:

```
HAMLET : packed array [1..17] of CHAR
```

Переменная HAMLET — *массив литер.* Массив литер — это упакованный массив, компоненты которого имеют тип CHAR и тип индекса имеет нижнюю границу, равную 1. Переменной HAMLET теперь может быть присвоено значение — строка, подобная следующей:

```
HAMLET = 'Быть или не быть?'
```

Так как при описании было указано, что HAMLET должен содержать 17 литер, любое значение, присваиваемое HAMLET, должно содержать точно 17 литер. В противном случае возникнет ошибка несоответствия типов. Для того, чтобы поместить в HAMLET более короткую строку, нужно дополнить строку пробелами так, чтобы она имела надлежащую длину:

```
HAMLET : = 'Ах,бедный Йорик!'
```

Строка 'Ах,бедный Йорик!' содержит 16 позиций, поэтому справа к ней добавлен один пробел. Приведенные ниже операторы присваивания также верны:

```
HAMLET := '12345                '
HAMLET := '+−*/                 '
HAMLET := '                     '
HAMLET := 'Я с"ел яблоко        '
```

Четвертая строка демонстрирует исключительный случай: использование в строке знака апострофа. Так как знак апострофа используется в качестве ограничителя, строки, то при записи строки, подобной

```
'Я  с' ел      яблоко'
```

возникает неоднозначность ее толкования.

ЭВМ не всегда способна определить, какой из знаков апострофа определяет конец строки. Как можно видеть из гл. 3, язык Паскаль исключает эту неоднозначность, требуя, чтобы знаки апострофа внутри строки писались дважды, как в приведенном операторе присваивания. ЭВМ запомнит такую строку в правильном виде, с одним знаком апострофа.

Для ввода массива литер могут быть использованы операторы READ и READLN. После того как массив литер получил значение, он может быть напечатан с помощью WRITE или WRITELN. (Массив литер — это единственный структурированный тип, который может обрабатываться подобным образом.) Приведенный ниже оператор напечатает значение переменной HAMLET без обрамляющих апострофов:

```
WRITELN(HAMLET)
```

Поскольку массивы литер являются обычными массивами, но их компоненты имеют тип CHAR, они обладают всеми свойствами регулярных массивов. Можно

получить копию всего массива с помощью одного оператора присваивания, а можно и извлечь значение отдельной литеры с помощью индексов массива. В приведенном цикле содержимое HAMLET печатается в обратном порядке:

```
for COUNT := 17 downto 1 do
      WRITE(HAMLET[COUNT])
```

### 10.3. Тип данных STRING

Наряду с тем положительным, что дают нам массивы литер, они обладают существенным недостатком: их длину нельзя изменить во время выполнения программы. В большинстве же программ приходится иметь дело со строками, длину которых определить заранее невозможно. По этой причине в языке Паскаль предусмотрен тип данных STRING. Переменная типа STRING в отличие от обычного массива литер может принимать значения переменной длины. Приведем несколько описаний, которые создают строковые переменные:

```
LONG_ LINE : STRING[200]
CITY : STRING[30]
OWNER : STRING
```

Каждая строковая переменная имеет атрибут длины, который определяет ее максимальную длину. Строковой переменной может быть присвоено любое значение *атрибута длины* до 255. Если в описании не указан атрибут длины, то по умолчанию переменной присваивается атрибут длины 80. В первых двух из приведенных выше примеров атрибут был записан явно. LONG_ LINE может иметь любую длину от 0 до 200; CITY может иметь любую длину от 0 до 30. В третьем примере атрибут был опущен. Следовательно, OWNER имеет максимальную длину 80. Некоторые операторы присваивания для OWNER будут выглядеть так:

```
OWNER := 'Смит'
OWNER := 'О"Коннор'
OWNER := ''
```

Эти три оператора присваивания задают переменной OWNER длины 4, 8 и 0 соответственно. Строка, представленная в третьем операторе присваивания, называется *пустой строкой:* она не содержит литер. Пустую

строку не следует путать со строкой, содержащей единичный знак апострофа. Для того чтобы поместить в OWNER знак апострофа, нужно записать так:

```
OWNER := ''''
```

Текущая длина строковой переменной может быть определена с помощью встроенной функции LENGTH. Для заданного значения типа STRING функция LENGTH возвращает целое значение, показывающее количество литер в строковом значении. Ниже приведен цикл, в котором печатается посимвольно содержимое OWNER:

```
for COUNT := 1 to LENGTH(OWNER) do
      WRITE (OWNER[COUNT])
```

Конечно, подобные циклы не нужно использовать в реальных программах. Так же как и литерные массивы, переменные типа STRING могут быть напечатаны с помощью единственного оператора WRITE или WRITELN. Отсутствие длины поля строковой переменной означает, что используется ее текущая длина. Если длина поля определена, ЭВМ сравнивает ее с текущей длиной строки. Если длина поля больше длины строки, происходит выравнивание строки по правому краю поля, а слева строка дополняется пробелами. Если поле меньше, чем строка, произойдет усечение строки справа так, чтобы она поместилась в поле. Если переменная TEN_DIGITS равна '0123456789', то

```
WRITELN(TEN_ DIGITS)      напечатает 0123456789
WRITELN(TEN_ DIGITS : 10) напечатает 0123456789
WRITELN(TEN_ DIGITS : 13) напечатает 0123456789
WRITELN(TEN_ DIGITS : 5)  напечатает 01234
```

Для того чтобы ввести значение типа STRING, необходимо использовать READLN, а не READ. Более того, за один раз может быть введена только одна строка. Приведем пример цикла, в котором вводится и печатается последовательность строк, пока не будет обнаружена строка 'STOP':

```
repeat
      READLN(LINE_OF_TEXT);
      WRITELN(LINE_OF_TEXT)
until LINE_OF_TEXT = 'STOP'
```

Две строки можно сравнивать, используя булевы операции отношения. Относительное упорядочивание строк выполняется аналогично упорядочиванию слов в словаре. Сначала сравниваются самые левые литеры; если они равны, то в сравнение вовлекаются следующие литеры и т. д. Приведенные ниже булевы выражения верны:

```
'Tommy' > 'Harry'
'France' = 'Francé
'Miscount' <= 'Misquote'
'Rockies' <> 'Alps'
'king' < 'kingdom'
```

Сравнение может дать неожиданные результаты, если в двух строках перемешаны заглавные и строчные буквы. Как видно из приложения В, у строчных букв в коде ASCII бóльшие порядковые номера, чем заглавные. Это означает, что строка 'pascal' больше, чем 'Pascal', которая в свою очередь больше, чем 'PASCAL'. При сравнении также могут получаться неожиданные результаты, если две строки содержат цифровые литеры. Так как сравнение строк производится в «словарном стиле», нельзя ожидать того, что сравнение строк будет соответствовать понятиям сравнения чисел. Например, несмотря на то, что 300 больше, чем 32, строка '300' на самом деле меньше, чем строка '32', так как '2' больше, чем '0'.

В примере 10.3 приведена программа, которая запоминает последовательность строк в массиве, компоненты которого имеют тип STRING. Вместо того чтобы сначала упорядочить компоненты массива, а затем их напечатать, эта программа просматривает массив каждый раз, когда печатает значение. Она начинает с поиска наименьшей строки массива, не принимая во внимание пустые строки. Найденная наименьшая строка распечатывается. В позицию наименьшей строки заносится пустая строка, так что она не будет повторно выводиться на печать, когда программа вновь будет просматривать массив для печати наименьшей строки. Программа заканчивается, когда не остается строк для печати.

Символические константы, так же как и переменные, могут представляться литерными строками. Подобно константам типов REAL, INTEGER или BOOLEAN, константа типа STRING имеет фиксированное значение,

**Пример 10.3. Распечатка строк в алфавитном порядке**

```
program ALPHABETIZE;
(*Эта программа читает список литерных строк и затем
печатает их в упорядоченном по алфавиту виде.*)
const
      LIST_SIZE = 10;
var
      COUNT, LIST_ENTRY : INTEGER;
      LOW_STRING : 1..LIST_SIZE;
      LINES : array[1..LIST_SIZE] of STRING;
begin
      (*--Чтение списка--*)
      WRITELN('Пожалуйста, введите ', LIST_SIZE,'
                  строк текста');
      WRITELN('которые нужно упорядочить по алфави-
                  ту. При каждом вводе задается значе-
                  ние одной строки.');
      for LIST_ENTRY := 1 to LIST_SIZE do
          READLN(LINES[LIST_ENTRY]);
      (*--Печать списка--*)
      for LIST_ENTRY := 1 to LIST_SIZE do
          begin
                LOW_STRING := 1;
                for COUNT := 2 to LIST_SIZE do
                    if (LINES[COUNT] < LINES[LOW_
                                          STRING]) and
                       (LINES[COUNT] <> '') then
                       LOW_STRING := COUNT;
                WRITELN(LINES[LOW_STRING]);
                LINES[LOW_STRING] := ''
          end
end.
```

которое не может быть изменено программой. За исключением этого ограничения, над строковой константой могут быть выполнены любые операции, которые могут выполняться над строковыми переменными.

Вот несколько описаний строковых констант:

```
const
      WARNING = 'Превышены временные границы!';
      TITLE     = 'Сумма оплаты';
      ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVW
      XYZ';
```

## 10.4. Встроенные операции над строками

Для того чтобы облегчить обработку значений типа STRING, в язык Паскаль введены специальные возможности для манипулирования строками. Большинство этих возможностей можно реализовать короткой последовательностью операторов; однако существование их в качестве встроенных элементов языка позволяет программисту сосредоточивать свое внимание на более существенных проблемах. Имеется семь встроенных функций над строковыми данными, включая функцию LENGTH, рассмотренную выше.

• Функция CONCAT принимает в качестве аргумента список строк и сцепляет их. Сцепить строки — значит построить такую их комбинацию, которая представляет единую строку. Например, если сцепляются строки 'ABC' и 'XYZ', то должна быть получена строка 'ABCXYZ'. Сцепление строк '12', '34' и '56' даст строку '123456'. Вот несколько операторов, использующих CONCAT:

```
NUMBERS := CONCAT ('12', '34', '56')
FULL_NAME := CONCAT(FIRST_NAME, ' ', LAST_
NAME)
SENTENCE := CONCAT(SENTENCE, '.')
```

В каждом из этих операторов присваивания CONCAT сцепляет все значения, записанные в круглых скобках; значение результата CONCAT имеет тип STRING. Например, третий оператор присваивания добавляет точку к концу значения переменной SENTENCE и помещает результат в SENTENCE. В CONCAT может обрабатываться любое число строк или строковых выражений.

• Функция POS в качестве аргументов использует две строки и определяет, содержится ли первая строка во второй. Если ответ положителен, POS выдает целое число, показывающее номер самой левой позиции образа первой строки в пределах второй. Иначе POS возвращает 0. Так,

```
значение POS ('e', 'Hello') равно 2
значение POS ('write', 'typewrite') равно 5
значение POS ('VAR', 'VAR') равно 1
значение POS ('ss', 'Mississippu') равно 3
значение POS ('sole', 'Brevity is the soul of wit') равно 0
```

• Функция COPY получает в качестве аргумента строку, а результатом выполнения функции является подстрока. Подстрока — это просто фрагмент большей строки. Если имеется строка 'ABCDEF', то список возможных подстрок может включать в себя следующие подстроки: 'ABC', 'BC', 'CDEF', 'E'.

При использовании COPY нужно определить начало подстроки и ее длину. Если заданы строка и два целых числа *i* и *j* , то COPY формирует литерную подстроку длиной *j*, начинающуюся с позиции *i* исходной строки. Для того чтобы напечатать первые три литеры строковой переменной ARTIST, следует написать

```
WRITELN (COPY (ARTIST, 1, 3))
```

Для того чтобы напечать последние пять литер переменной ARTIST, следует написать

```
WRITELN (COPY (ARTIST, LENGTH (ARTIST) - 4, 5))
```

Ниже приведен вложенный цикл, с помощью которого печатаются все подстроки переменной ARTIST, включая и ее самое:

```
for START := 1 to LENGTH(ARTIST) do
        for SIZE := 1 to LENGTH(ARTIST) - START
        + 1 do
                WRITELN(COPY(ARTIST, START,
                SIZE))
```

• Оператор DELETE можно воспринимать как обратный по отношению к COPY. Если заданы строка и два целых числа, DELETE удаляет указанную подстроку из строки. Если START равен 3, а SIZE равен 2, то DELETE (NAME, START, SIZE) удалит третью и четвертую литеры из NAME. Приведенный пример демонстрирует использование оператора DELETE:

```
QUOTATION := 'резвая коричневая лиса';
DELETE(QUOTATION, 8, 11);
WRITELN(QUOTATION);
```

Последний оператор WRITELN напечатает: 'резвая лиса'. Отметим, что DELETE — это оператор, а не функция. Функцию оператора DELETE можно смоделировать с помощью CONCAT и COPY. Например, оператор

```
DELETE(QUOTATION, 8, 11)
```

эквивалентен последовательности

```
QUOTATION := CONCAT(COPY(QUOTATION, 1,7),
                    COPY(QUOTATION, 19, LENGTH
                    (QUOTATION) - 18))
```

• Оператор INSERT помещает одну строку внутрь другой. Если START равен 12, то INSERT(INITIAL, NAME, START) вставит содержимое INITIAL в NAME, начиная с NAME [12]. Приведем последовательность операторов, в которой демонстрируется использование INSERT:

```
WORD := 'коричневая';
QUOTATION := 'резвая лиса';
INSERT(WORD, QUOTATION, 8);
WRITELN(QUOTATION);
```

Оператор WRITELN напечатает: 'резвая коричневая лиса'. INSERT, как и DELETE, может быть заменен с помощью CONCAT и COPY. Оператор

```
INSERT(WORD, QUOTATION, 8)
```

эквивалентен последовательности

```
QUOTATION := CONCAT(COPY(QUOTATION, 1,7),
                    WORD
                    COPY(QUOTATION, 8, LENGTH
                    (QUOTATION) - 7))
```

Программа, показанная в примере 10.4, считывает строку и затем разбивает ее на последовательность слов. Поскольку предполагается, что слова разделены пробелами, используется POS, чтобы определить положения границ слова. Каждое слово, которое выбирается из начала OLD_LINE, добавляется в начало NEW_LINE, что приводит к изменению порядка слов исходной строки. Когда в OLD_LINE не остается слов, содержимое NEW_LINE печатается.

• Оператор STR имеет целочисленный аргумент и выдает в качестве результата строку. Если, например, RATE равно 120, то STR(LINE, RATE) устанавливает LINE равным '120'. В программах, осуществляющих взаимные преобразования строковых и

**Пример 10.4. Изменение порядка слов в строке**

```
program REVERSE_WORDS;
(*Эта программа читает строку, содержащую последо-
вательность слов, разделенных пробелами. Затем слова
печатаются в порядке, обратном по отношению к по-
рядку их следования в строке.*)
var
    OLD_LINE, NEW_LINE : STRING;
    BLANK : INTEGER;
    WORD : STRING;
begin
    NEW_LINE := '';
    WRITELN('Пожалуйста, введите строку, слова ко-
            торой должны быть реверсированы:');
    READLN(OLD_LINE);
    OLD_LINE := CONCAT(OLD_LINE,' '); (*добавить пробел*)
    while OLD_LINE <> '' do
        begin
            BLANK := POS(' ', OLD_LINE);
            WORD := COPY(OLD_LINE, 1,
                                BLANK);
            NEW_LINE := CONCAT(WORD,
                                NEW_LINE);
            DELETE(OLD_LINE, 1, BLANK)
        end;
    WRITELN;
    WRITELN('Вот строка с реверсированным порядком
            слов:');
    WRITELN(NEW_LINE)
end.
```

целочисленных значений, гораздо чаще выполняется обратное преобразование — из строки в целое. Однако в языке Паскаль подобная возможность в виде встроенного средства отсутствует. Преобразовать строковую переменную NUMBER в целочисленную и поместить результат в качестве значения VALUE можно следующим образом:

```
VALUE := 0;
for DIGIT := 1 to LENGTH(NUMBER) do
    VALUE := 10*VALUE + ORD(NUMBER[DIGIT])
                                - ORD('0');
```

После проверки очередной цифры VALUE «сдвигается» влево с помощью умножения, и новая цифра добавляется к VALUE. Ниже приводится более универсальная версия такого преобразования, которая проверяет наличие в первой позиции знак плюс (+) или минус (–).

```
VALUE := 0;
MINUS := FALSE;
if (NUMBER[1] = '+') or (NUMBER[1] = '–') then
    begin
        MINUS := (NUMBER[1] = '–');
                                    (* отрицательное? *)
        DELETE(NUMBER, 1,1)
    end;
for DIGIT := 1 to LENGTH(NUMBER) do
    VALUE := 10 * VALUE + ORD(NUMBER[DIGIT]) –
                                        – ORD('0');
if MINUS then
    VALUE := –VALUE;
```

Первый оператор *if* проверяет, присутствует ли знак. Если знак присутствует, то MINUS принимает значение TRUE для знака минус и FALSE для знака плюс. Затем литера знака удаляется из строки (если знак отсутствует, то это означает, что, конечно, MINUS равен FALSE).

ВОПРОСЫ

*1. Напишите программу, которая принимает на входе строку, содержащую последовательность заглавных и строчных букв, затем печатает эту строку только литерами заглавных букв.

2. Напишите две программы, которые получают на входе строку, содержащую значение римской цифры от 1 до 10 (т. е. от I до X), и печатают ее десятичный эквивалент. Одна программа должна использовать оператор *case*, другая должна обрабатывать строку литера за литерой.

*3. Корректны ли строки, значениями которых являются зарезервированные слова (например, 'BEGIN' или 'VAR')?

4. Чему равно значение ORD('XYZ')?

5. *Палиндром* — это такая строка, значение которой не изменяется при реверсировании ее литер (знаки пунктуации и пробелы при сравнении не учитываются). Следующие строки являются примерами палиндромов:

```
'OTTO'
'MADAM, I"M ADAM'
'A MAN, A PLAN, A CANAL - - PANAMA'
```

Напишите программу, которая проверяет, является ли строка палиндромом.

6. Напишите цикл *for*, который воспроизводит результат выполнения следующего оператора:

```
NEW_NAME := COPY(OLD_NAME, START, SIZE)
```

*7. Пусть имеются следующие описания:

```
const
      HEADER = 'Отчет об успеваемости студентов';
var
      SCHOOL_NAME : STRING;
      STUDENT_NAME : STRING[25];
```

LENGTH(SCHOOL_NAME) равна 34. LENGTH (STUDENT_NAME) равна 11. Какие из приведенных операторов неверны?

а) INSERT(STUDENT_NAME, SCHOOL_NAME, 1)
б) DELETE(SCHOOL_NAME, LENGTH(SCHOOL_NAME), 1)
в) INSERT('*', HEADER, 3)
г) STUDENT_NAME := CONCAT(STUDENT_NAME, 'посещает'; SCHOOL_NAME)
д) SCHOOL_NAME := COPY(HEADER, 1,5)
е) STUDENT_NAME := SCHOOL_NAME[10]

# Глава 11

# Описание функций, определяемых программистом

## 11.1. Описание функций

Язык Паскаль обладает широким набором встроенных функций. В это множество входят такие арифметические функции, как ABS и SQRT; функции преобразования типа, как, например, TRUNC, и функции ввода-вывода, например EOF. Несмотря на это, в языке Паскаль могут отсутствовать некоторые функции, весьма полезные в конкретной программе. Не существует языка, содержащего все средства, которые могли бы потребоваться программисту, и язык Паскаль — не исключение из этого правила. Язык Паскаль обладает механизмом описания функций, что дает возможность программисту дополнить язык нужными ему средствами.

Предположим, что в некоторой программе необходимо выполнить возведение в степень. Средства возведения в степень языка Паскаль ограничиваются двумя стандартными функциями SQR и PWR. Однако может возникнуть необходимость выполнить возведение с другим показателем степени. Фактически нужна функция, которая позволяет вычислять выражения, подобные этим:

$3.24^{3}$
$1.414^{-2}$
$17^{0}$

Программист может решить ввести функцию, называемую POWER. Эта функция должна иметь два аргумента: целочисленный и вещественный. Таким образом, с использованием функции POWER приведенные выше выражения можно записать в следующем виде:

POWER (3.14,3) = $3.14^{3}$ = 30.9591
POWER (1.414,−2) = $1.414^{-2}$ = 0.5
POWER (17.0) = $17^{0}$ = 1.0

Пример 11.1 показывает образец, как можно описать функцию POWER. После описания функция POWER

используется аналогично стандартным функциям. В программе описание функции должно располагаться между описанием переменных и телом программы.

**Пример 11.1. Возведение в степень**

```
function POWER(FACTOR : REAL; EXPONENT :
                                  : INTEGER) : REAL;
(*Эта функция возводит вещественное число в целочисленную степень.*)
var
    COUNT : INTEGER;
    TEMPFACTOR : REAL;
begin
    if EXPONENT = 0 then
        POWER := 1
    else
        begin
            TEMPFACTOR := FACTOR;
            for COUNT := 2 to ABS(EXPONENT) do
                TEMPFACTOR := TEMPFACTOR
                * FACTOR;
            if  EXPONENT < 0 then
                POWER := 1 / TEMPFACTOR
            else
                POWER := TEMPFACTOR
        end
end; (*функции POWER*)
```

**Заголовок функции**

Общая форма записи:

function *имя (список параметров): тип результата*
function *имя: тип результата*

*Тип результата* представляется идентификатором типа, который описывает тип значения, получаемого после выполнения функции.

*Список параметров* строится из параметров данной функции. Если список параметров не определен, то функции никакие аргументы не передаются.

*Примеры*

```
function ARCTAN(THETA : REAL) : REAL
function ADDCOLORS(COLOR1, COLOR2 : HUE) : HUE
function TSCORE(DEGREES : INTEGER; SIGNIF :
        REAL) : REAL
function DEVICEREADY : BOOLEAN
```

Первая строка описания является заголовком функции. Заголовок функции определяет имя функции, число и тип параметров, а также тип значения, получаемого в результате выполнения функции. Имя функции является некоторым идентификатором, и это имя должно удовлетворять всем требованиям, которые предъявляются к идентификаторам языка Паскаль. Имя должно начинаться с буквы, состоять только из букв и цифр и не совпадать с зарезервированными словами. Ниже представлены некорректные заголовки функций:

```
function 90 DEGREE(ANGLE : REAL) : REAL
function DIGIT? (NEXT : CHAR) : BOOLEAN
function BEGIN(ROW : INTEGER) : CHAR
```

Элементы, проводимые в скобках, образуют параметры функции. POWER имеет два параметра: FACTOR (типа REAL) и EXPONENT (типа INTEGER). При обращении к функции POWER, FACTOR и EXPONENT принимают значения соответствующих аргументов. В теле функции FACTOR и EXPONENT используются аналогично обычным переменным. В следующих операторах присваивания происходит обращение к функции POWER:

```
HYPOTENUSE := SQRT(POWER(A, 2) + POWER(B, 2))
VOLUME := POWER(LENGTH, 3)
RANK := 2 + POWER(SCORE / 2, TRUNC(OLDRANK))
```

Каждый аргумент должен иметь тот же тип, что и соответствующий ему параметр в заголовке функции. Когда аргумент представляется выражением, как в третьем из приведенных операторов присваивания, то, прежде чем передать управление функции, выполняется вычисление выражения. Следовательно, если значение SCORE равно 4.0 и значение OLDRANK равно 3.5, то FACTOR и EXPONENT будут соответственно равны 2.0 и 3.

Тип результата функции является либо порядковым типом, либо типом REAL. Результат функции не может быть массивом или другим структурированным типом. В рамках этих ограничений выбор типа результата представляется достаточно свободным. Это может быть стандартный тип, такой, как REAL или BOOLEAN, или ранее определенный тип. В любом случае тип

результата должен определяться идентификатором типа. Приведенные ниже заголовки неверны, так как тип результата задается не идентификатором типа:

```
function SEASON(THISMONTH : MONTHS) :
          (WINTER, SPRING, SUMMER, FALL)
function DAYSINMONTH(THISMONTH : MONTHS) :
          1..31
```

Выбор типа параметра представляется еще более свободным. В качестве типа параметра может быть любой тип, включая структурированный. Так же как и для результата, для определения типа параметра следует использовать идентификатор типа. (Помните, что описание типа используется для образования идентификатора вновь создаваемого типа.)

Если несколько параметров имеют один и тот же тип, то можно сократить список параметров, сгруппировав эти параметры. Особенно важно группировать вместе те параметры, которые логически связаны между собой. Ниже приведен пример двух форм записи заголовка одной и той же функции:

```
function AZIMUTH(ANGLE1 : REAL; ANGLE2 : REAL;
          LENGTH1: INTEGER; LENGTH2:
          INTEGER) : REAL
function AZIMUTH(ANGLE1, ANGLE2 : REAL;
          LENGTH1, LENGTH2 : INTEGER) :
          REAL
```

Следующим после заголовка элементом описания функции является раздел описаний (*внутреннее описание*). Функция POWER имеет внутреннее описание двух переменных: COUNT и TEMPFACTOR. Переменные, описанные в рамках функции, недоступны в основной программе. Это временные переменные, которые используются только в пределах самой функции. COUNT используется как счетчик цикла, а TEMPFACTOR — для хранения результатов умножений. После завершения выполнения функции POWER переменные COUNT и TEMPFACTOR исчезнут и появятся вновь при очередном вызове функции.

Последним элементом описания функции является тело функции. Тело функции состоит из операторов, выполнение которых приводит к вычислению резуль-

тата функции. В теле функции может быть использован любой корректный оператор языка Паскаль. Для указания значения, которое является результатом выполнения функции, следует использовать оператор присваивания с идентификатором функции в левой части. Например, следующий оператор присваивания приводит к тому, что результат выполнения функции равен 1:

```
POWER := 1
```

В теле функции выполняется проверка переменной EXPONENT на равенство нулю. Если равенство выполняется, то результат выполнения функции POWER равен 1, так как любое число в нулевой степени равно 1. Если переменная EXPONENT не равна нулю, то возведение в степень выполняется с использованием переменной TEMPFACTOR. После завершения вычислений значение TEMPFACTOR присваивается POWER и таким образом устанавливается результат выполнения функции.

Функция MEAN (пример 11.2) представляет пример функции, аргументом которой является массив. LISTSIZE представляет константу, значение которой задано в основной программе. Если тип LIST описан в основной программе для обозначения некоторого массива, компоненты которого имеют тип REAL (или INTEGER) и индексы которого имеют тип 1..LISTSIZE, то эта функция вычисляет среднее значение компонент массива.

---

**Пример 11.2. Среднее значение компонент массива**

```
function MEAN(SAMPLES : LIST) : REAL;
(*Эта функция вычисляет среднее значение чисел, содержащихся в массиве SAMPLES.*)
var
      COUNT : INTEGER;
      SUM : REAL;
begin
      SUM := 0;
      for COUNT := 1 to LISTSIZE do
            SUM := SUM + SAMPLES[COUNT];
      MEAN := SUM / LISTSIZE
end; (*функции MEAN*)
```

---

## 11.2. Локальные и глобальные идентификаторы

Программа, состоящая из набора функций, представляется рядом *концентрических блоков.* Программа в примере 11.3 состоит из внешнего блока — самой программы — и внутреннего по отношению к программе блока. Внутренний блок представляет функцию с именем UPPER.

---

**Пример 11.3. Преобразование к заглавным буквам**

```
program CHANGECASE;
(*Эта программа вводит текстовую строку и меняет все строчные буквы на заглавные.*)
var
    LINE : STRING;
    COUNT : INTEGER;
function UPPER(ONECHAR : CHAR) : CHAR;
(*Эта функция получает одну литеру. Если она является строчной буквой, то возвращается заглавный эквивалент этой буквы. В противном случае возвращается первоначальная литера.*)
const
    UPPERCASE = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
    LOWERCASE = 'abcdefghijklmnopqrstuvwxyz';
var
    LETTER : INTEGER;
begin
    LETTER := POS(ONECHAR, LOWERCASE);
    if LETTER = 0 then
        UPPER := ONECHAR
    else
        UPPER := UPPERCASE[LETTER]
end; (*функции UPPER*)
begin (*основная программа*)
    WRITELN('Введите, пожалуйста, строку для преобразования к заглавным буквам.');
    READLN(LINE);
    for COUNT := 1 to LENGTH(LINE) do
        LINE[COUNT] := UPPER(LINE[COUNT]);
    WRITELN;
    WRITELN('Преобразованная строка');
    WRITELN(LINE)
end.
```

---

Некоторые из идентификаторов, включая LETTER, ONECHAR, UPPERCASE и LOWERCASE, описаны в рамках функции UPPER. Поскольку эти идентификаторы описаны для данной функции, то и доступны они только в пределах данной функции. Другими словами, действие описания идентификаторов распространяется лишь в пределах наименьшего блока, содержащего это описание. Такие идентификаторы называют *локальными* по отношению к UPPER.

С другой стороны, некоторые идентификаторы определены в разделе описаний основной программы. К таким идентификаторам относятся LINE и COUNT. Описание идентификаторов также действует в рамках блока, содержащего эти описания. Так как в данном случае блок представляет всю программу, идентификаторами можно пользоваться в любом месте программы. Эти идентификаторы можно применять не только в теле основной программы, но также в пределах действия описаний функции UPPER. Поэтому такие идентификаторы называют *глобальными* по отношению к UPPER.

Различие между локальными и глобальными идентификаторами является существенным. Поведение локальной переменной совершенно отлично от поведения глобальной. Переменная, которая является локальной по отношению к некоторой функции, создается при каждом вызове функции. После того как выполнение функции завершается, переменная уничтожается. Таким образом, локальная переменная недоступна вне блока, в котором она описана.

Правила, регулирующие действие идентификаторов внутри программных блоков, называются *правилами локализации* (области действия). Областью действия является та область, в которой определен данный идентификатор. Спрашивая: «Какова область действия X?» — мы в действительности хотим узнать: «В каких частях программы мы можем использовать X?» Тем самым мы подошли к первому правилу локализации.

- **Первое правило локализации**

*Идентификатор определен только в пределах блока, в котором он описан.*

Идентификаторы, описанные в разделе описаний основной программы, принадлежат к так называемым *глобальным* идентификаторам. LINE и COUNT являются

**Пример 11.4. Программа, которая содержит описания нескольких функций**

```
program ANYNAME;
const
      NAMELENGTH = 20;
var
      STATUS : BOOLEAN;
      FIELD : INTEGER;
      USERNAME : packed array[1 .. NAMELENGTH]
      of CHAR;
          function ENTRY(ONECHAR : CHAR) :
          INTEGER;
          var
                  COUNT : INTEGER;
                      function SEARCH(POSITION :
                      INTEGER) : BOOLEAN;
                      var
                        INDEX : 1 .. NAMELENGTH;
                      begin
                      .

                      .
                      end; (* функции SEARCH *)
            begin
                    .
                    .
                    .
            end; (* функции ENTRY *)
            function BACKTRACK(FINALPOS :
                     INTEGER) : INTEGER;
            const
                 BASE = 7;
            var
                 TRANSFORM : packed array
                 [1.. NAMELENGTH] of CHAR;
            begin
                        .
                        .
                        .
          end; (* функции BACKTRACK *)
begin (* Основная программа *)
    .
    .
    .
end.
```

примером глобальных идентификаторов. Так как глобальные идентификаторы описываются в самом внешнем блоке, они доступны из любого места программы. Например, переменной LINE может присваиваться некоторое значение или она может быть распечатана в пределах описания функции UPPER.

В примере 11.4 приведен скелет программы, в которой содержатся три функции: ENTRY, SEARCH и BACKTRACK. Используем эту программу для демонстрации действия первого правила в более сложном случае. Отметим, что идентификаторы NAMELENGTH, STATUS, FIELD и USERNAME являются глобальными. Следовательно, эти идентификаторы можно использовать в любом месте программы.

Идентификаторы ONECHAR и COUNT являются локальными идентификаторами функции ENTRY. Эта функция в свою очередь содержит в своем составе функцию SEARCH. Идентификаторы POSITION и INDEX являются локальными идентификаторами функции SEARCH. В операторах функции SEARCH возможно использование ONECHAR, COUNT, а также глобальных идентификаторов. (Например, константа NAMELENGTH используется в качестве верхней границы интервала в описании переменной INDEX.)

FINALPOS, BASE и TRANSFORM являются локальными идентификаторами функции BACKTRACK. В результате эти идентификаторы недоступны за пределами BACKTRACK. Нельзя использовать эти идентификаторы в операторах функции ENTRY или в теле основной программы.

Предположим, что возникла необходимость описать один и тот же идентификатор дважды: один раз в функции ENTRY и другой в функции BACKTRACK. Можно ли сделать это? Да, можно. Идентификатор может иметь различное описание в каждом отдельном блоке. Этот идентификатор в действительности может представлять два различных объекта программирования. Так, в пределах ENTRY идентификатор представляет один объект программирования, а в пределах BACKTRACK этот же идентификатор олицетворяет уже другой объект. Таким образом, следующее правило локализации можно сформулировать так:

• **Второе правило локализации**

*Один и тот же идентификатор может быть поразному определен в каждом отдельном блоке.*

Это вовсе не значит, что к такому дублированию идентификаторов надо стремиться, наоборот, следует по возможности избегать такого дублирования, так как это приводит к затруднениям при чтении программы. Едва ли упрощается задача понимания сложной программы, если скажем, некоторый идентификатор в одном блоке представляет литерную константу, а в другой — тип данных.

Приняв во внимание данный совет, рассмотрим другую проблему: возможно ли переопределить некоторый идентификатор во вложенном блоке? Другими словами, если идентификатор COUNT описан в функции ENTRY, то можно ли под этим идентификатором описать что-то другое в функции SEARCH? Ответ на этот вопрос положительный. Идентификатор в таких случаях интерпретируется в соответствии с третьим правилом локализации:

• **Третье правило локализации**

*Если в некотором операторе используется идентификатор, который описан в ряде концентрических блоков, то такой идентификатор интерпретируется в соответствии с описанием самого внутреннего из вложенных блоков, содержащих данный оператор.*

Программа, приведенная в примере 11.5, содержит идентификатор COUNT как в основной программе, так и в функции ISPRIME. В каждом из блоков имя COUNT является целочисленной переменной. Тем не менее эти переменные отличаются. В рамках ISPRIME идентификатор COUNT представляет локальную переменную, а за пределами COUNT — глобальную переменную. Если бы ISPRIME содержала внутреннюю функцию, то эта функция могла бы содержать еще один уровень определения идентификатора COUNT.

Четвертое и последнее правило локализации представляет уже известное правило, которое заключается в том, что все идентификаторы в блоке должны быть уникальными. Один и тот же идентификатор может описываться в независимых блоках или во вложенных блоках, но в одном блоке не может быть нескольких

**Пример 11.5. Программа с одинаковыми именами переменных**

```
program PRIMETABLE;
(*Эта программа печатает список чисел, указывая, явля-
  ется ли число простым или нет.*)
var
    COUNT : INTEGER;
    START, FINISH : INTEGER;
        function ISPRIME(NUMBER : INTEGER) :
                BOOLEAN;
        (*Эта функция определяет, является ли число
          простым.*)
        var
            COUNT : INTEGER;
        begin
            ISPRIME := TRUE;
            for COUNT := 2 to NUMBER - 1 do
                if (NUMBER mod COUNT) = 0 then
                    ISPRIME := FALSE
        end; (* функции ISPRIME *)
begin
    WRITELN('С какого числа должен начинаться спи-
                                                сок?');
    READLN(START);
    WRITELN('Каким числом должен оканчиваться спи-
                                                сок?');
    READLN(FINISH);
    for COUNT := START to FINISH do
        begin
            WRITE(COUNT);
            if ISPRIME(COUNT) then
                WRITELN('простое число.')
            else
                WRITELN('не простое число.')
        end
end.
```

описаний одного идентификатора. Например, глобальная переменная не может иметь такого же имени, как и функция. Нельзя также, чтобы перечисляемый элемент имел такое же имя, как и символическая константа.

• **Четвертое правило локализации**

*На уровне блока идентификатор может описываться лишь один раз.*

Третье правило локализации касается и стандартных идентификаторов языка Паскаль. В частности, в отношении правил локализации стандартные идентификаторы не отличаются от идентификаторов, вводимых программистом. Можно трактовать стандартные идентификаторы, исходя из модели, согласно которой любая программа располагается в некотором воображаемом блоке, в котором описаны все стандартные идентификаторы. Такая модель выглядела бы следующим образом:

```
const
      MAXINT = 32767;
type
      INTEGER = -MAXINT..MAXINT;
      BOOLEAN = (FALSE, TRUE);
      .
      .
      .
function SIN(ANGLE : REAL) : REAL;
      .
      .
      .
function ROUND(VALUE : REAL) : INTEGER;
      .
      .
      .
function ODD(VALUE : INTEGER) : BOOLEAN;
      .
      .
      .
      program ANYNAME;
      .
      .
      .
      end.
```

Хотя такая точка зрения представляет чисто технический интерес, она имеет неожиданное следствие: точно так же, как на более низком уровне возможно

переопределить глобальный идентификатор, в программе имеется возможность изменить интерпретацию стандартного идентификатора. Излишне говорить, что это не практично и не должно поощряться.

## 11.3. Остерегайтесь побочных эффектов

С математической точки зрения функция выполняет только одно действие: по значению аргументов вычисляет результат функции. Однако в языке Паскаль работа функции связана с выполнением ряда других действий. Например, функция может изменить значение глобальной переменной или распечатать какие-нибудь данные. Эти дополнительные действия называются *побочными эффектами* функции.

В идеале функцию следовало бы рассматривать как «черный ящик», который на входе получает набор аргументов и по ним формирует единственный выход, называемый *результатом функции.* Действительно, было бы удобно вызвать функцию с уверенностью, что это не приведет к каким-то другим действиям. Такой взгляд на функцию является предпочтительным в том плане, что он избавляет программиста от необходимости вникать в работу функции.

Побочные эффекты приводят к тому, что функция ведет себя совершенно по-другому. Функцию с побочными эффектами уже нельзя представить в виде «черного ящика» с четко определенными входами и единственным выходом для результата. Вместо этого функция становится творением со все менее предсказуемым поведением, что требует от программиста постоянного внимания к деталям реализации. С этой точки зрения старайтесь не закладывать побочных эффектов при написании функции.

Возможны случаи непреднамеренного создания побочных эффектов. В примере 11.6 имя ANGLE является одновременно и глобальной, и локальной переменной. В теле основной программы эта переменная используется как указатель цикла, а в рамках DISTANCE – как промежуточная переменная. Программа в той форме, как она представлена, будет приводить к ошибке, так как программист забыл описать ANGLE в рамках DISTANCE. Поэтому каждое обращение к

**Пример 11.6. Непреднамеренный побочный эффект**

```
program MISTAKE;
(*Эта программа формирует таблицу, показывающую
  расстояние до объекта, если известны высота и угол
  возвышения.*)
var
    ANGLE, ALTITUDE : REAL;
function DISTANCE(ELEVATION, HEIGHT : REAL) :
        REAL;
const
    PI = 3.14159;
begin
    ANGLE := ELEVATION * PI / 180; (*перевод в ра-
                                          дианы *)
    DISTANCE := HEIGHT / (SIN(ANGLE) / COS
                                          (ANGLE))
end; (*функции DISTANCE *)
begin
    WRITELN('На какой высоте объект?');
    READLN(ALTITUDE);
    WRITELN(' ');
    WRITELN('Угол возвышения' : 20,  'Расстояние' : 10);
    ANGLE := 0;
    While ANGLE <= 90 do
          begin
              WRITELN(ANGLE : 20 : 3, DISTANCE
              (ANGLE, ALTITUDE) : 10 : 3);
              ANGLE := ANGLE + 0.5
          end
end.
```

ANGLE есть обращение к глобальной переменной. Это означает, что при каждом вызове функции DISTANCE будет иметь место неприятный побочный эффект — будет меняться переменная цикла.

## 11.4. Рекурсивные функции

К функции можно обращаться тремя способами: из тела основной программы, из тела другой функции, из тела самой функции. Первые два способа не представляют ничего неожиданного. Однако третий способ

может вызвать некоторое замешательство: как функция может вызывать сама себя?

В ряде случаев такие функции не являются чем-то исключительным. Функции, которые вызывают сами себя, называются *рекурсивными.*

Некоторые математические функции наиболее просто выражаются в рекурсивной форме. Факториал X (записывается как X!) является произведением целых чисел от 1 до X. Если не использовать рекурсивного представления, то X! равен X * (X − 1) * (X − 2)...*3 * 2 * 1. В рекурсивной форме определение выглядит следующим образом:

для X > 0, X! = X * (X − 1)!
для X = 0, X! = 1

Программа, приведенная в примере 11.7, использует описанный алгоритм. FACTORIAL является рекурсивной функцией, которая вызывает сама себя до тех пор, пока не будет выполнено условие NUMBER = 0.

При каждом рекурсивном вызове создается новое множество локальных переменных. Таким образом, рекурсивный вызов не изменяет переменные, расположенные вне вызываемой функции. Если при первоначальном вызове FACTORIAL значение аргумента равно 5, то последовательность вызовов и результаты будут выглядеть следующим образом:

```
FACTORIAL(5) = 5 * FACTORIAL(4)
             = 5 * 4 * FACTORIAL(3)
             = 5 * 4 * 3 * FACTORIAL(2)
             = 5 * 4 * 3 * 2 * FACTORIAL : (1)
             = 5 * 4 * 3 * 2 * 1 * FACTORIAL(0)
             = 5 * 4 * 3 * 2 * 1 * 1
             = 120
```

---

**Пример 11.7. Рекурсивные вычисления факториалов**

```
program FINDFACTORIAL;
(*Эта программа вычисляет значение факториала, используя рекурсивную функцию.*)
var
    NUMBER : INTEGER; (* вводимое число *)
function FACTORIAL(VALUE : INTEGER) : INTEGER;
begin
```

```
    if VALUE = 0 then
        FACTORIAL := 1
    else
        FACTORIAL := VALUE * FACTORIAL
        (VALUE - 1)
end;    (* функции FACTORIAL *)
begin
    WRITELN('Факториал какого числа следует вы-
                                    числить?');
    READLN(NUMBER);
    if NUMBER < 0 then
        WRITELN('Отрицательные числа не имеют
                                    факториала.')
    else
        WRITELN('Факториал', NUMBER, 'равен',
        FACTORIAL(NUMBER))
end.
```

Показательная функция является примером другой функции, которая легко приводится к рекурсивной форме. В примере 11.8 показан рекурсивный вариант функции POWER. Оба описания этой функции обладают одной особенностью, которая является характерным свойством рекурсивных функций: они проверяют значение аргументов для принятия решения о *завершении.* Для функции FACTORIAL условие завершения VALUE = 0. Для функции POWER условие завершения EXPONENT = = 0. Следующие друг за другом вызовы рекурсивной

**Пример 11.8. Рекурсивный вариант вычисления экспонент**

```
function POWER(FACTOR : REAL; EXPONENT :
        INTEGER) : REAL;
begin
    if EXPONENT < 0 THEN
        POWER := 1 / POWER(FACTOR, ABS
        (EXPONENT))
    else
        if EXPONENT > 0 then
            POWER := FACTOR * POWER(FACTOR,
            EXPONENT - 1)
        else
            POWER := 1
end; (* функции POWER *)
```

функции должны формировать аргументы, которые обеспечивают постепенную сходимость к условию завершения.

Другой класс рекурсивных функций составляют функции, выполняющие *косвенную рекурсию.* Косвенная рекурсия возникает в том случае, когда некоторая функция обращается к другой функции, а та в свою очередь вызывает первоначальную функцию. Для предотвращения бесконечной последовательности вызывов одна или обе функции должны проверять условие завершения. Однако функциям данного типа присуща так называемая *проблема ссылок вперед.*

По ряду технических причин идентификаторы в программах на языке Паскаль не могут появляться прежде, чем они будут описаны. Это означает, что функция не может выполнить обращение к другой функции, описание которой располагается в этой программе ниже. Если описание функции ARCSINE предшествует описанию функции INNERPRODUCT и оба блока находятся на одном уровне влажности, то INNERPRODUCT может вызывать ARCSINE, но ARCSINE не может вызвать INNERPRODUCT. Пример 11.9 демонстрирует данную проблему для косвенной рекурсии.

При косвенной рекурсии обязательно возникает ссылка вперед. (В рассмотренном случае функция ARCSINE делает ссылку вперед к INNERPRODUCT.) Язык Паскаль обеспечивает возможность косвенной рекурсии при помощи выполнения предварительного описания, которое предшествует обычному описанию. Предварительное описание состоит из заголовка функции, за которым следует слово FORWARD[1]. Пример 11.10 показывает, как может быть выполнена косвенная рекурсия между ARCSINE и INNERPRODUCT с использованием предварительного описания INNERPRODUCT.

В приведенном примере ARCSINE может вызывать INNERPRODUCT, несмотря на то, что полное описание функции INNERPRODUCT по-прежнему располагается

---

[1] Предварительное указание называется *директивой.* Предварительное описание состоит из заголовка функции, за которым следует директива FORWARD. В последующем описании функции опускаются список формальных параметров и тип результата. Предварительное описание и последующее описание функции должны быть локальны в одном и том же блоке и образуют описание функции на месте предварительного описания. — *Прим. ред.*

в программе ниже. Отметим также, что список параметров и тип функции INNERPRODUCT представлены только в предварительном описании и не повторяются в полном описании функции.

---

**Пример 11.9. Некорректная преждевременная ссылка**

```
function ARCSINE(SINEVALUE : REAL) : REAL;
var
    THETA : REAL;
    LEG1, LEG2 : VECTOR;
begin
    .
    .
    .
THETA := INNERPRODUCT(LEG1, LEG2);
    .
    .
    .
end; (* функции ARCSINE *)
function INNERPRODUCT(ALPHA, BETA : VECTOR)
        : REAL;
var
    NUMERATOR : REAL;
begin
    .
    .
    .
NUMERATOR := ARCSINE(BETA[2]);
    .
    .
    .
end; (* функции INNERPRODUCT *)
```

---

---

**Пример 11.10. Косвенная рекурсия с использованием предварительного описания**

```
function INNERPRODUCT(ALPHA, BETTA : VECTOR)
                        : REAL; FORWARD;
function ARCSINE(SINEVALUE : REAL) : REAL;
    THETA : REAL;
    LEG1, LEG2; VECTOR;
```

```
begin
  .
  .
  .
THETA := INNERPRODUCT(LEG1, LEG2);
  .
  .
  .
end; (* функции ARCSINE *)
function INNERPRODUCT;
var
    NUMERATOR : REAL;
begin
  .
  .
    NUMERATOR := ARCSINE(BETA[2]);
  .
  .
  .
end; (* функции INNERPRODUCT *)
```

---

ВОПРОСЫ

*1. Какие из приведенных заголовков функции являются верными?

```
function    NEXTDAY(TODAY : DAYSOFWEEK) : DAYS-
OFWEEK
function TAN (ANGLE : REAL) : REAL;
function SIGN (CONST : INTEGER) : INTEGER;
function MAXINT (INDEX : INTEGER) : INTEGER;
```

*2. Какие идентификаторы в примере 11.7 являются локальными для FACTORIAL?

*3. Напишите описание функции ISDIGIT, которая получает на входе одну литеру и вырабатывает результат TRUE, если литера является цифрой, и FALSE в противном случае.

4. Напишите описание функции PRIME, которая получает на входе целое число и вырабатывает результат TRUE, если это число простое, и FALSE в противном случае.

5. Если I и J — положительные целые числа, то функция Аккерманна определяется следующим образом:

для I = 0, ACK (I, J) = J + 1
для J = 0, ACK (I, J) = ACK (I−1,1)
в противном случае ACK (I, J) = ACK (I−1, ACK (J−1))

Напишите описание рекурсивной функции, которая вычисляет ACK (I, J).

6. Напишите описание функции, которая вычисляет факториалы, не прибегая к рекурсии.

# Глава 12

# Описание процедур, определяемых программистом

## 12.1. Описание процедур

Наряду с тем что язык Паскаль предоставляет механизмы поддержки создаваемых программистом функций, он позволяет также программисту определять новые процедуры. *Процедура* — это подпрограмма, которая решает некоторую частную задачу. Встроенные процедуры языка Паскаль включает в себя WRITE, WRITELN, READ, READLN, Новые процедуры описываются способом, подобным способу описания нестандартных функций. В примере 12.1 приведено описание процедуры, определяемой программистом.

---

**Пример 12.1. Описание процедуры**

```
procedure PRINTBARS(BARS : INTEGER);
var
      COUNT : INTEGER;
begin
      for COUNT := 1 to BARS do
       WRITE ('-')
end; (* процедуры PRINTBARS *)
```

---

PRINTBARS — это процедура с аргументом типа INTEGER. Отметим, что в заголовке процедуры в отличие от заголовка функции не определяется тип результата. Эта разница возникает из-за того, что процедура не возвращает формального результата; задача процедуры — выполнить некоторые действия.

Каждый раз, когда программа обращается к процедуре PRINTBARS, будет выполняться распечатка строки подчеркиваний. Аргумент, передаваемый с помощью BARS, определяет число печатаемых подчеркиваний. Описав процедуру PRINTBARS, ее можно использовать так же, как и обычный оператор языка Паскаль.

Приведем пример *for* — цикла, который использует PRINTBARS, чтобы распечатать образ треугольника:

```
for LENGTH := 1 to 5 do
    begin
        PRINTBARS(LENGTH);
        WRITELN
    end
```

Оператор WRITELN используется, чтобы перевести позицию печати на начало новой строки. Результирующая выдача будет выглядеть так:

```
_
_ _
_ _ _
_ _ _ _
_ _ _ _ _
```

В процедуре может быть произвольное число аргументов; на тип аргумента не накладывается никаких ограничений. В процедурах действуют те же правила области действия, которые применяются в функциях. Описания процедур в программе помещаются вместе с описаниями функций. Приведем для справки сведения о порядке расположения описаний в программе.

---

**Структура блока**

Общая форма:

**Заголовок**
**Описания меток**
**Определения констант**
**Описания типов**
**Описания переменных**
**Описания функций и процедур**
**Тело**

Элементы блока должны появляться в этом порядке. За исключением заголовка и тела, любой элемент может быть опущен. Например, если не используются операторы *goto*, в секции описания меток нет необходимости.

---

Процедуры и функции, имеющие внутренние описания, должны следовать тем же правилам расположения описаний, что и основная программа. Вообще процедуру или функцию нужно воспринимать как программу в миниатюре. В них могут быть собственные переменные, собственные описания типов и даже собственные функции и процедуры.

Процедура примера 12.2 содержит внутреннюю функцию ISALPHA. Процедура использует ISALPHA,

чтобы определить, является ли литера из CHUNK буквой. Если это так, то под ней печатается черточка, т. е. оператор

UNDERLINE ('Это — строка.')

напечатает:

Это — строка.

Процедура может вызывать себя рекурсивно. Подобно рекурсивной функции, рекурсивная процедура должна включать в себя механизм завершения рекурсии, чтобы предотвратить бесконечное зацикливание. Некоторые приложения рекурсивных процедур мы рассмотрим в гл. 17.

---

**Пример 12.2. Процедура с внутренней функцией**

```
procedure UNDERLINE(CHUNK : STRING);
var
    POSITION : INTEGER;
        function ISALPHA(ONECHAR : CHAR)
                BOOLEAN;
        const
            ALPHABET = 'ABCDEFGHIJKLMNOPQ-
            RSTUVWXYZ';

        var
            LETTER : INTEGER;
        begin
            ISALPHA := FALSE;
            for LETTER := 1 to 26 do
                if ONECHAR = ALPHABET
                  [LETTER] then
                  ISALPHA := TRUE
        end; (* функции ISALPHA *)
begin
    WRITELN(CHUNK);
    for POSITION := 1 to LENGTH(CHUNK) do
            if ISALPHA(CHUNK[POSITION]) then
                WRITE('_')
end; (* процедуры UNDERLINE *)
```

---

## 12.2. Параметры-переменные

Иногда использование функций может оказаться совершенно недостаточным. Результатом выполнения функции является некоторое значение. Функции не могут быть использованы в ситуациях, когда результат представляется двумя величинами или массивом. Путем использования в таких случаях процедур вместо функций можно создавать подпрограммы, которые возвращают столько значений, сколько необходимо.

Один из способов получения в качестве результата нескольких значений заключается в использовании процедур, которые возвращают результаты, изменяя глобальные переменные. Однако тот же эффект может быть получен с использованием параметров-переменных (вызов по наименованию). В примере 12.3 приведена процедура, которая использует параметр-переменную, чтобы возвратить в качестве результата массив.

Идентификатор *var,* предшествующий LINE в списке параметров, показывает, что LINE является параметром-переменной. Параметр-переменная отличается от обычного параметра (т. е. параметра-значения) тем, что все присваивания, которые делаются параметру-переменной, делаются и соответствующему аргументу. Если USERNAME является переменной типа STRING, то следующий вызов процедуры приведет к сжатию содержимого USERNAME:

```
COMPRESS (USERNAME)
```

После вызова процедуры USERNAME будет изменен в соответствии с алгоритмом преобразования LINE внутри процедуры. Если USERNAME равен 'ABCDE', то вызов COMPRESS приведет к тому, что значение USERNAME будет 'ABCDE'. Так как LINE — параметр-переменная, то аргумент, соответствующий LINE, должен быть переменной. Ясно, что процедура не может передать значение через выражение или константу. По этой причине приведенный ниже вызов процедуры содержит ошибку:

```
COMPRESS ('ABCDE')
```

**Пример 12.3. Процедура, которая сжимает строку**

```
procedure COMPRESS(var LINE : STRING);
(*Эта процедура удаляет все пробелы из LINE.*)
var
    BLANK_SPOT : INTEGER;
begin
    BLANK_SPOT := POS(' ',LINE);
    while BLANK_SPOT <> 0 do
        begin
          DELETE(LINE, BLANK_SPOT, 1);
          BLANK_SPOT := POS('  ', LINE)
        end
end; (* процедуры COMPRESS *)
```

Если процедура вырабатывает свой результат, то она передает его вызванной программе только через параметр-переменную. В языке Паскаль каждый вызов процедуры или функции проверяется на соответствие аргументов соответствующему списку параметров; если обнаруживается расхождение, пользователь получает предупреждающее сообщение. В списке параметров может быть определено любое сочетание параметров-значений и параметров-переменных. Приведем еще несколько заголовков процедур, в которых появляются параметры-переменные:

```
procedure REVERSE (var LINE : STRING;
                   var PALINDROME : BOOLEAN)
procedure TRANSPOSE(var GAMMA : MATRIX)
procedure MOVECURSOR(X, Y : INTEGER;
                     var SECTOR : INTEGER)
```

В первом примере определяется два параметра-переменных: первый типа STRING и второй типа BOOLEAN. Во втором примере, как и в COMPRESS, определяется один параметр-переменная определенного программистом типа данных массива. В третьем примере за двумя параметрами-значениями определяется единственный параметр-переменная.

Существует одно важное ограничение на использование параметра-переменной. В качестве параметра-переменной не могут передаваться компоненты массива. Это правило влечет за собой некоторые аномалии. Так,

поскольку строка — это упакованный массив типа CHAR, единичная литера строки не может быть передана как параметр-переменная. (Кроме того, встроенные процедуры READ и READLN используют параметры-переменные. Следовательно, их нельзя использовать, чтобы прочитать значение в единичный компонент упакованного массива.)

## 12.3. Еще раз о передаче управления

Как отмечалось в разд. 7.4, необходимость передачи управления возникает достаточно редко, так как управляющие структуры языка Паскаль в большинстве ситуаций позволяют избежать использования этого оператора. Одной из причин, по которой передачи управления могут быть нужны, — это необходимость преднамеренно прекратить выполнение программы. Например, можно использовать оператор перехода для прекращения дальнейшей обработки при получении неправильных входных данных. (Даже эта причина, кстати, не является абсолютной, так как обычно существуют более предпочтительные реакции на неправильные данные. Если ничего нельзя сделать, то существует возможность предложить пользователю проверить данные или ввести их более аккуратно.)

Передачи управления из тела функции или процедуры связаны с дополнительными сложностями. В отличие от идентификаторов, которые могут определяться в различных блоках в соответствии с правилами локализации, описанными в разд. 11.2, метки определены только в теле блока, где расположено их описание. Соответственно для перехода в блок или из блока оператор *goto* не может быть использован.

---

**EXIT**
**Общая форма записи:**
EXIT *(имя программы)*
EXIT *(имя функции)*
EXIT *(имя процедуры)*
EXIT *(program)*

**Если аргументом EXIT является имя функции или процедуры, то выполнение указанной функции или процедуры завершается. Управление передается в вызвавший блок. Если аргументом EXIT является имя программы или зарезервированное слово *program*, то выполнение программы завершается.**

***Примеры:***
EXIT (program)
EXIT (UNDERLINE)
EXIT (ISALPHA)

---

Для того чтобы выполнить выход из блока, мы используем процедуру EXIT. EXIT позволяет передать управление любому внешнему блоку, в том числе и выйти из программы. Предположим, мы находимся внутри функции IS_DIGIT, которая является внутренней по отношению к функции CONVERT, а последняя является внутренней по отношению к процедуре READ_NUMBER. Вызов EXIT (IS_DIGIT) приведет к возврату в ту точку CONVERT, откуда происходил вызов IS_DIGIT. Аналогично EXIT (CONVERT) приведет к возврату в ту точку READ_NUMBER, откуда происходил вызов CONVERT.

Если EXIT выполняется в пределах функции или процедуры, которые вызывались рекурсивно, то управление возвращается на предыдущий уровень рекурсии. Важно заметить, что если EXIT обрабатывается в пределах функции прежде, чем определен результат ее выполнения, это вызывает ошибку вида «неопределенное значение». Это замечание относится и к параметру-переменной. Если переменная, переданная как параметр-переменная, была не определена до вызова процедуры и ей все еще не присвоено значение в момент выполнения EXIT, то и после завершения вызова процедуры она остается неопределенной.

## 12.4. Преимущества модульности

Не случайно функции и процедуры похожи на программы в миниатюре. Язык Паскаль спроектирован таким образом, что подпрограммы могут рассматриваться как независимые объекты, работа которых не зависит от основной программы. Как следствие наличие функций и процедур позволяет проектировать и конструировать программы модульным, структурированным способом.

Наиболее важное преимущество модульности, даваемое подпрограммами, — это возможность уменьшить число повторений одной и той же последовательностей операторов. В очень больших или средних по размеру программах существует по крайней мере несколько процессов, выполнение которых повторяется в различных частях программы. Было бы неразумно в программе выполняющей тригонометрические преобразования, вычислять ARCSIN в десятках различных мест. Описание функции — гораздо более простое решение.

Однако использование подпрограмм вызывается не только соображениями краткости. Кроме уменьшения числа повторений последовательности операторов применение подпрограмм позволяет также в пределах программы отделить задачи одну от другой. Такое выделение задач ценно по нескольким причинам. Во-первых, это увеличивает наглядность и, следовательно, понимание программы. Если тело программы представляет собой набор операторов для решения задач, то читателю трудно получить общее представление о работе программы. В противоположность этому если программа сконструирована как набор отдельных подпрограмм, она становится понятнее, так как ее логическая структура более связана с физическим смыслом.

Во-вторых, выделение задач с помощью подпрограмм является также эффективным средством разработки программ. При разработке больших или средних по размерам программ на программиста сразу обрушивается целый поток деталей проектирования.

Использование подпрограмм делает процесс программирования более систематическим и регулируемым по нарастанию сложности. Благодаря использованию подпрограмм программист может рассматривать программу как набор более мелких компонентов, которые проектируются отдельно.

Учитывая сложность программ, этот эффект является более полезным, чем может показаться на первый взгляд. Как правило, сложность программ возрастает в геометрической прогрессии с ее размером, так что разбиение программы на меньшие части действительно уменьшает усилия, необходимые для ее разработки. Использование подпрограмм позволяет программисту минимизировать сложность программы с помощью применения стратегии «разделяй и властвуй». Начиная с рудиментарного описания функций программы, можно постепенно разбивать программу на отдельные задачи, которые в свою очередь могут быть реализованы как объединение функций и процедур.

Как видим, использование функций и процедур является естественным дополнением к технике программирования, позволяющим создавать качественные программы. Выделение задач улучшает наглядность программ, однако разговор на этом не заканчивается. Используя

продуманные названия для функций и процедур, программист может выработать более наглядный стиль написания программ. В программе примера 12.4 проил-

---

**Пример 12.4. Модульная программа обработки статистики**

```
program STATS;
const
      MAXSIZE = 100;
type
      LIST : array[1 .. MAXSIZE] of REAL;
var
      LISTSIZE : INTEGER;
      LOWER, UPPER : REAL;
      DATA : LIST;
procedure READLIST(var NUMBERS : LIST);
var COUNT : INTEGER;
begin
      WRITELN('Сколько элементов в списке?');
      READLN(LISTSIZE);
      WRITELN('Введите, пожалуйста, элементы списка:');
      for COUNT := 1 to LISTSIZE do
            READLN(NUMBER[COUNT])
end; (* процедуры READLIST *)
function MEAN(NUMBERS : LIST) : REAL;
var
      SUM : REAL;
      ELEMENT : INTEGER;
begin
      SUM := 0;
      for ELEMENT := 1 to LISTSIZE do
            SUM := SUM + NUMBERS[ELEMENT];
      MEAN := SUM / LISTSIZE
end; (* функции MEAN *)
function VARIANCE(NUMBERS : LIST) : REAL;
var
      AVERAGE  := REAL;
      COUNT : INTEGER;
      SUM : REAL;
begin
      SUM := 0;
      AVERAGE := MEAN(NUMBERS);
      for COUNT := 1 to LISTSIZE do
            SUM := SUM + SQR(NUMBERS[COUNT] -
            AVERAGE) / LISTSIZE;
```

```
      VARIANCE := SUM
end; (* функции VARIANCE *)
procedure FINDRANGE(NUMBERS : LIST; var MIN,
                                          MAX : REAL);
var   COUNT : INTEGER;
begin
      MIN := NUMBERS[1];
      MAX := NUMBERS[1];
      for COUNT := 2 to LISTSIZE do
            begin
              if NUMBERS[COUNT] < MIN then
          MIN := NUMBERS[COUNT];
              if NUMBERS[COUNT] > MAX then
                MAX := NUMBERS[COUNT]
end
end; (* процедуры FINDRANGE *)
begin (* тело программы *)
      READLIST(DATA);
      WRITELN('Среднее   значение   равно ', MEAN
                                              (DATA));
     WRITELN('Дисперсия равна ', VARIANCE(DATA));
      FINDRANGE(DATA, LOWER, UPPER);
      WRITELN('Диапазон  значений  от  ', LOWER,
                                        ' до ', UPPER)
end.
```

---

люстрировано применение такого подхода. Каждая задача в STATS оформлена как отдельная подпрограмма. Каждой подпрограмме присвоено имя со смысловой нагрузкой, так что основная программа STATS становится понятной при минимальных справках о подпрограммах. В идеале тело основной программы должно иметь вид описания, в котором приводится информация о том, что́ программа делает, а не как делает. Детали алгоритма реализуются в отдельных функциях и процедурах.

ВОПРОСЫ

*1. Напишите описание процедуры SWAP, которая получает два аргумента типа REAL и меняет их значения местами.

2. Напишите описание процедуры REAL_STRING, которая получает значение типа REAL и возвращает две строки: одна представляет вещественное число

в обычном виде, а другая — в показательной форме.

3. Напишите описание функции STRING_REAL, которая получает строку, представляющую вещественное число (в обычной или показательной форме), и возвращает ее вещественный эквивалент.

*4. Вначале определите тип данных для представления квадратичного полинома (квадратичный полином имеет вид $aX^2 + bX + C$).

Затем напишите процедуру, которая получает два линейных полинома и вычисляет их произведение.

5. Напишите описание процедуры PRINTPOLY, получающей полином, представленный типом данных, описанным в задаче 4. Процедура должна напечатать этот полином.

# Глава 13
# Важность надежности программ

## 13.1. Почему программы допускают ошибки?

Часто приходится сталкиваться с фактами, заставляющими начать сомневаться в способности ЭВМ давать точные результаты. Существует бесчисленное количество достоверных историй о том, как ЭВМ посылала напоминания о необходимости уплатить 0.00 долл. или выписывала платежные чеки на сумму, выраженную семизначным числом. Такие ошибки совершенно недопустимы.

Явление, которое мы называем «ошибка ЭВМ», чаще всего не является следствием неправильной работы вычислительной машины. В большинстве вычислительных систем (за исключением тех, которые были созданы еще в эпоху вакуумных ламп) неправильные результаты нельзя объяснить отказом оборудования. Более того, «ошибка ЭВМ» в действительности является программной ошибкой. Почему так происходит?

Частично ответ на этот вопрос состоит в следующем: программы ошибаются потому, что допускают ошибки люди, которые их пишут. Программы содержат ошибки в значительной степени по тем же причинам, по которым, например, в деловом письме может оказаться неправильно написанное слово. В обоих случаях конечный продукт есть результат написания; в обоих случаях ошибка есть следствие человеческой неаккуратности.

Однако этот ответ оставляет часть проблемы нерешенной: почему процесс программирования более сильно подвержен ошибкам, чем другие процессы человеческой деятельности, которые, по крайней мере на первый взгляд, одинаково восприимчивы к способности человека ошибаться? Если книга дважды проверена квалифицированным корректором, можно с большой долей уверенности сказать, что эта книга не содержит ошибок. Программу для ЭВМ нельзя проверить так же просто. Ошибки в программе обнаруживаются через

месяцы и даже через годы после номинального завершения разработки программы.

Можно смело предположить, что программист не глупее авторов монографий. Трудность составления надежной программы состоит в том, что программист должен учесть мириады взаимодействий. Каждое входное значение непредсказуемо; если бы его можно было предсказать, то зачем его вводить? Далее, каждое отдельное значение входной переменной приводит к тому, что другие переменные могут принимать различные значения, а это в свою очередь приводит к изменению порядка выполнения операторов и т. д. до бесконечности.

Сказанное выше заставляет сделать вывод: при проектировании программы следует ставить в качестве цели обеспечение надежности. Увеличение сложности взаимодействий создает условия для того, чтобы программист совершал ошибки; следовательно, усилия, направленные на уменьшение сложности, должны привести к уменьшению количества ошибок. Учитывая, что сложность является основным детерминантом надежности, при выработке собственного стиля программирования нужно стремиться создавать условия для повышения достоверности программ. Как следствие отработанная методика программирования является тем средством, при помощи которого можно с минимальными затратами добиться повышения надежности.

Достичь этого можно простой адаптацией приемов программирования, позволяющих создавать хорошие программы. К счастью, шансы сделать ошибку уменьшаются, если при разработке программы стремиться к тому, чтобы она была понятна другим людям. Как было отмечено ранее, программы, написанные с учетом этих позиций, и легче разрабатывать, так как достигается простота понимания отдельных деталей проекта. Таким образом, хороший программист помогает не только тем, кто будет читать и использовать его программы, но и себе самому.

## 13.2. Тестирование

Даже если вы в высшей степени аккуратны в написании программы, нужно быть готовым лицом к лицу встретиться с неприятной истиной: скорее всего, при первом запуске программа работать не будет. Случается,

что программа начинает работать сразу, при первом запуске, и тем не менее это не опровергает аксиомы. Последнее утверждение может показаться чрезмерно пессимистичным, но опытные программисты знают, что оно просто констатирует факт.

После того как завершены проектирование и написание программы, работа программиста еще не закончена. Если он хочет использовать свою программу для достаточно серьезных целей, он должен предположить, что она содержит ошибки. Сделав это предположение, программист должен постараться найти как можно больше ошибок. Процесс поиска ошибок в программе называется *тестированием*. Назначением тестирования является проведение «строгого судилища» над программой, прежде чем она будет отдана в эксплуатацию.

Для того чтобы оттестировать программу, нужно подобрать представительный набор контрольных примеров. *Контрольный пример* — это такое сочетание значений входных переменных, которое может быть использовано для получения информации о надежности проверяемой программы. Если прогон контрольного примера приводит к отказу программы, значит, мы обнаружили ошибку в программе. Программист при этом ставится в своеобразное положение, поскольку он должен подобрать такие контрольные примеры, которые бы наилучшим образом «угробили» его программу. Контрольный пример, который программа обрабатывает правильно, менее значим, чем тот, который приводит к неверной работе, поскольку последний дает больше информации о недостатках программы.

Тестирование программы должно включать в себя прогон трех видов контрольных примеров: *нормальные ситуации, граничные ситуации* и *случаи неправильных данных*. Для того чтобы пояснить смысл этих терминов, рассмотрим тривиальный пример проектирования контрольных примеров. Допустим, у нас есть программа, которая решает квадратное уравнение, задаваемое в форме $aX^2 + bX + c = 0$. Какие значения A, B и C нужно выбрать для контрольных примеров?

Мы начнем с серии нормальных случаев. Они «нормальны» в том смысле, что представляют собой примеры правильных входных данных. Если программа не работает в подобных случаях, она требует очень серьезных переделок. Вот несколько примеров нормаль-

ных ситуаций для программы, решающей квадратные уравнения:

A = 1, B = 2, C = 1
A = 3, B = 7, C = 3
A = 4, B = 5, C = 0

Если мы удовлетворены прогоном нормальных ситуаций, нужно переходить к проверке граничных ситуаций. Граничные контрольные примеры помогают установить, способна ли программа нормально реагировать на особые случаи во входных данных. Граничные примеры представляют собой данные, которые, будучи математически корректными, приводят программу к необходимости работать особым образом. Приводимые ниже контрольные примеры являются граничными (почему?):

A = 5, B = 4, C = 3
A = 0, B = 10, C = 7
A = 0, B = 0, C = 0

Напомним, что корни считаются по формуле $(-b \pm \sqrt{b^2-4AC})/2A$. Если приведенные выше значения коэффициентов будут использоваться для вычисления по формуле, будет получен неверный результат. Первое множество значений приведет к необходимости извлекать корень из отрицательного числа; второе и третье вызовут ошибку из-за деления на ноль. И все-таки все три примера представляют правильные входные данные. Первое выражение, $5X^2+4X+3$, имеет комплексные корни. Второе, $10X+7$, является линейным полиномом с корнем $X=7/10$. Третье, $0X^2+0X+0$, имеет бесконечное множество корней.

После того как исследованы граничные случаи, необходимо подготовить последовательность примеров с неправильными данными. Неправильными являются такие данные, которые расположены вне допустимого диапазона. Случай A = 0, B = 0, C = 1 для нашей гипотетической программы является примером с неправильными данными, так как ему соответствует «уравнение» 1 = 0, которое, конечно, неверно. Примеры с неправильными данными должны быть обработаны соответствующим образом, поскольку в повседневной эксплуатации программе придется иметь дело с неверными входными данными.

## 13.3. Отладка

Обнаружив ошибку, мы сталкиваемся с необходимостью исправить ее. Можно использовать ряд различных методов отладки, позволяющих обнаружить расположение ошибки; выбор «наилучшего» существенно зависит от особенностей ситуации. Большинство программистов начинают с неформального метода, известного под названием *проверка за столом.*

Проверка за столом представляет собой вид корректуры, при котором программист претендует на роль ЭВМ. Используя контрольный пример, который привел к ошибке в программе, программист аналитически трассирует листинг программы в надежде локализовать ошибку. Проверка за столом — это хороший метод, поскольку он заставляет программиста детально понять работу программы.

К сожалению, действенность этого метода уменьшается из-за склонности человека «читать между строк». Читая свой собственный листинг, программист обычно видит то, что он хочет видеть, даже если в программном листинге написано совершенно другое. Очень просто, например, может быть пропущена ошибка перфорации (типа SQR вместо SQRT). Тенденция читать то, что хочется, очень сильна, и существует единственное противоядие: настройка на скрупулезное внимание к деталям.

Если использование метода проверки за столом оказалось бесплодным, нужно обратиться к более изощренным методам. Несмотря на то что специфика таких методов меняется от программы к программе, почти всегда привлекаются отчеты о *трассировке.* Трассировка — это процесс указания последовательности выполнения операторов. Для того чтобы получить отчет о трассировке, включают временно операторв WRITELN в стратегические (ключевые) точки по всей программе. Например, если в программе содержится процедура с именем NEXTLINE, в описание тела процедуры может быть включен следующий оператор:

```
WRITELN ('**Вызвана NEXTLINE.**')
```

Если NEXTLINE вызывалась слишком часто (или недостаточно часто), эта трассировка сделает ошибку более заметной.

Для того чтобы сделать трассировку более информативной, она может быть спроектирована так, чтобы показать историю изменения некоторых выбранных переменных. Например:
WRITELN ('**NEXTLINE, COUNT=', COUNT,' и KEY = ', KEY).

Каждый раз, когда вызывается NEXTLINE, печатаются текущие значения COUNT и KEY. Таким образом мы можем наблюдать за потоком управления и за значением важных переменных. Результирующая выдача позволит локализовать подозрительные ситуации.

Как только ошибка найдена, ее нужно исправить и затем продолжить поиск других ошибок. Однако не следует слишком спешить в этот момент, так как часто исправление ошибок порождает множество новых ошибок. С другой стороны, исправление может ликвидировать лишь часть причин, оставив неизменными остальные. Чтобы предотвратить обе ситуации, после того, как ошибки исправлены, необходимо провести новое тестирование программы.

## 13.4. Подпрограммы и изоляция ошибок

Поскольку надежность программы тесно переплетается с ее сложностью, можно предположить, что использование подпрограмм может помочь максимизировать надежность. Это действительно так. Поскольку использование функций и подпрограмм позволяет разрабатывать программу более систематизированно, то это позволяет также и более систематизированно проводить тестирование.

Ключ к систематизации тестирования — в изоляции ошибок. На любой стадии процесса разработки хотелось бы иметь по крайней мере скелет программы в основном свободным от ошибок. Добившись этого, мы могли бы при каждом наращивании программы изолировать новые ошибки в ее новых частях.

Для того чтобы в процессе разработки использовать изоляцию ошибок, необходимо рассматривать программу как последовательность промежуточных программ; каждая промежуточная программа реализует прогрессивно нарастающую часть желаемого продукта. Другими словами, спецификации большой программы

должны быть разбиты на такую последовательность программ, что каждая из них представляет собой хорошо определенное подмножество следующей. Когда сделано очередное добавление, наступает очередной раунд тестирования. При этом все ошибки могут быть локализованы и исправлены, прежде чем начнутся работы по очередному добавлению.

В такой схеме очень хорошо применим взгляд на подпрограммы как на «черные ящики». Если программа сконструирована как набор подпрограмм, каждая из которых решает единственную задачу, то изоляция ошибок может быть легко достигнута. Подпрограмма должна использоваться в качестве естественной единицы приращения при разбиении процесса разработки. Начиная с небольшого числа функций и процедур и скелета тела программы, мы можем постепенно добавлять по одной подпрограмме, предварительно выполнив ее тестирование. Теперь, если ошибка обнаружена, мы можем предполагать, что она находится в одном из двух мест: в самой новой подпрограмме или в том месте, где эта подпрограмма вызывается.

# Глава 14
# Множества

## 14.1. Описание множеств

До сих пор рассматривался лишь один структурированный тип языка Паскаль — массив. Однако в языке Паскаль имеются и другие структурированные типы, к которым принадлежит и тип данных множество. Математическое понятие множества подразумевает совокупность элементов. Например, множество четных целых чисел от 0 до 10 состоит из

{0, 2, 4, 6, 8, 10}

Множество, состоящее из числа 73, выглядит следующим образом:

{73}

Аналогично формируются множества на языке Паскаль[1]. Только вместо фигурных скобок для представления множеств используются квадратные. Описание на языке Паскаль приведенных выше множеств будет выглядеть так:

```
[0, 2, 4, 6, 8, 10]
[73]
```

Подобно массивам, переменная типа множество имеет тип компонент. Каждое число, входящее в множество, имеет значение, которое должно принадлежать к типу компонент. Множество, тип компо-

[1] По определению тип множество задает интервал значений, который является множеством всех подмножеств (множеством-степенью) соответствующего базового типа. Поэтому, если описана переменная, имеющая тип 1...3, то она может принимать значения из множества {1, 2, 3}. Если же она имеет тип SET OF 1 ..3, то она может принимать значение из следующего множества: {{1, 2, 3}, {1, 2}, {1, 3}, {2, 3}, {1}, {2}, {3}, { }}. Заметим также, что элементы множеств не упорядочены, поэтому множества {1, 3, 5}, {5, 3, 1} и {3, 5, 1} одинаковы. — *Прим. ред.*

нент которого является, например, интервалом 0..20, может быть представлено одной из следующих совокупностей элементов:

[0, 3, 10, 20]
[4, 5, 6, 7]
[15]

Следующие наборы элементов для указанного множества являются недопустимыми:

[-1, -2, -3]
[0, 10, 20, 30]
[1.5, 1E+06, 6.3]

В рамке приведены правила описания структурированной переменной типа множество. Тип компонент множества в отличие от типа компонент массива должен быть порядковым. Невозможно, например, образовать множество, элементами которого являются массивы или вещественные числа.

---

**Описание структурированной переменной типа множество**
Общая форма записи:
*имя:* set of *тип компонент*
Тип компонент должен быть порядковым
*Примеры:*
LETTERS : set of 'A'..'Z'
COLORS : set of HUES
HOLIDAYS : set of 1..31
USEDCHARS : set of CHAR

---

Имея описание множества, ему можно присвоить некоторое значение. Предположим, что задано такое описание:

```
type
    DAYSOFWEEK = (SUN, MON, TUES, WED,
    THURS, FRI, SAT);
var
    WEEKDAYS, WEEKEND : set of DAYSOFWEEK
```

После такого описания допустимы присваивания, подобные этим:

```
WEEKDAYS := [MON, TUES, WED, THURS, FRI]
WEEKEND := [SAT SUN]
```

Первое присваивание можно записать короче, используя интервальную форму записи. Таким образом:

```
WEEKDAYS := [MON..FRI]
```

Иногда возникает необходимость записать множества, не содержащие элементов. Такие множества называются *пустыми* и представляются следующим образом: [ ]

В языке Паскаль обеспечен механизм для определения принадлежности некоторого значения множеству его элементов. Этот механизм реализуется булевым оператором *in*. Следующие булевы выражения истинны, если выполнены приведенные выше присваивания для WEEKDAYS и WEEKEND:

```
WED in WEEKDAYS
FRI in WEEKDAYS
SAT in WEEKEND
```

И как следствие следующие булевы выражения ложны:

```
SAT in WEEKDAYS
SUN in WEEKDAYS
TUES in WEEKEND
```

Предположим, что TODAY является переменной типа DAYSOFWEEK. Приведенный оператор *if* будет определять принадлежность представленного значения TODAY к элементам множества WEEKDAYS:

```
if TODAY in WEEKDAYS then
        WRITELN ('Сегодня рабочий день')
else
        WRITELN ('Сегодня день отдыха')
```

Оператор *in* совместно с другими операторами может составлять булевы выражения. Например:

```
if (TODAY in WEEKDAYS) and (TODAY <> FRI) then
        WRITELN ('Сегодня либо понедельник, либо вторник, либо среда, либо четверг.')
```

В примере 14.1 приводится процедура, которая в качестве параметра содержит множество. Идентификаторы LOWER и UPPER представляют символические константы (ранее описанные программистом), которые определяют верхнюю и нижнюю границы интервала. Предполагается, что SMALLSET является идентификатором типа, описываемого множеством из элементов интервала LOWER.. UPPER.

Если, скажем, константа LOWER равна 0, а UPPER

равна 10, то приведенные обращения к процедуре PRINTSET корректны:

---

**Пример 14.1. Распечатка элементов множества**

```
procedure PRINTSET(OUTPUTSET : SMALLSET);
(*Эта процедура распечатывает элементы множества
  OUTPUTSET. Предлагается, что SMALLSET имеет тип
  компонент LOWER .. UPPER. *)
var
    MEMBER : INTEGER;
begin
    for MEMBER := LOWER to UPPER do
        if MEMBER in OUTPUTSET then
            WRITELN(MEMBER)
end;      (* процедуры PRINTSET *)
```

---

```
PRINTSET ([5, 6, 7])
PRINTSET ([2])
PRINTSET ([1...4])
PRINTSET ([ ])
```

Если ROOMSIZES является переменной типа SMALLSET, то следующее обращение к процедуре также является корректным:

```
PRINTSET (ROOMSIZES)
```

Множества часто используют для упрощения сложных операторов *if*. Приведем два оператора *if*, которые проверяют одно и то же условие:

```
if (TEMPERATURE = 0) or (TEMPERATURE = 32)
    or (TEMPERATURE = 212)
    or (TEMPERATURE = 276) then ...
if TEMPERATURE in [0, 32, 212, 276] then
```

Хотя эти операторы эквивалентны, второй оператор значительно проще. В ряде ситуаций использование множеств позволяет улучшить наглядность и понимание алгоритма работы программы. Например, можно определить, является ли литерная переменная, именуемая ONECHAR, цифрой, записав

```
if ONECHAR in ['0'..'9'] then
```

Как вы, наверное, заметили, в этом разделе не использовались большие интервалы типа INTEGER. Это объясняется довольно просто: числовые множества не могут содержать значений, меньших 0 и больших 4079[1]. Принимая во внимание данное ограничение, нельзя выполнять описания множеств, подобных INTEGER или состоящих из интервала 0...MAXINT. Самое большое множество целых чисел в языке Паскаль представляется интервалом [0...4079].

## 14.2. Действия над множествами

Язык Паскаль предоставляет ряд средств для выполнения алгебраических операций над множествами. К таким операциям относятся: *объединение множеств, пересечение множеств* и *разность множеств.* Прежде чем перейти к иллюстрации каждой из операций, введем следующее описание:

```
type
    COUNTRIES = set of (ENGLAND, FRANCE,
                        GERMANY, SPAIN, ITALY)
```

• *Объединением* двух множеств является множество, составленное из элементов обоих множеств. Символом для обозначения операции объединения множеств принимается знак плюс (+). Таким образом:

```
[ENGLAND, FRANCE] + [ITALY] → [ENGLAND, FRANCE, ITALY]
[SPAIN] + [GERMANY] → [SPAIN, GERMANY]
[FRANCE, SPAIN] + [ENGLAND, ITALY] → [ENGLAND,
        FRANCE, SPAIN, ITALY]
[FRANCE, GERMANY] + [ ] → [FRANCE, GERMANY]
[FRANCE] + [GERMANY] + [SPAIN] → [FRANCE, GERMANY, SPAIN]
[ITALY] + [ITALY, ENGLAND] → [ITALY, ENGLAND]
[SPAIN] + [SPAIN] → [SPAIN]
```

• *Пересечением* двух множеств является множество, состоящее из элементов, которые входят одновременно

---

[1] Заметим, что описываемая здесь реализация очень «щедрая». В большинстве реализаций число элементов в множестве лежит между 64 и 256. — *Прим. ред.*

в оба множества. В качестве символа для обозначения операции пересечения множеств принимается знак звездочка (*). Таким образом:

```
[ENGLAND, FRANCE, GERMANY] * [ENGLAND, GERMANY] → [ENGLAND, GERMANY]
[ITALY, SPAIN] * [ENGLAND, SPAIN, FRANCE] → [SPAIN]
[ITALY, SPAIN] * [GERMANY] → [ ]
[FRANCE] * [FRANCE] → [FRANCE]
[ENGLAND, SPAIN, GERMANY] * [ITALY, SPAIN, FRANCE, ENGLAND] → [SPAIN, ENGLAND]
[ITALY] * [ ] → [ ]
[ENGLAND..ITALY] * [FRANCE, SPAIN] → [FRANCE, SPAIN]
[ ] * [ ] → [ ]
```

• *Разностью* двух множеств является множество, состоящее из элементов первого множества, которые не являются вместе с тем элементами второго множества. В качестве символа для обозначения операции разности двух множеств принимается знак минус (—). Таким образом:

```
[ENGLAND..ITALY] - [FRANCE..SPAIN] → [ENGLAND, ITALY]
[ENGLAND..ITALY] - [ITALY, SPAIN] → [ENGLAND, FRANCE, GERMANY]
[ENGLAND, FRANCE, SPAIN] - [FRANCE] → [ENGLAND, SPAIN]
[ENGLAND, FRANCE, GERMANY] - [FRANCE, SPAIN] → [ENGLAND, GERMANY]
[GERMANY..ITALY] - [ENGLAND..SPAIN] → [ITALY]
[ENGLAND, FRANCE] - [ ] → [ENGLAND, FRANCE]
[ ] - [ENGLAND..ITALY] → [ ]
[SPAIN] - [SPAIN] → [ ]
```

Эти три оператора используются для построения выражений над множествами. Например, если MAP1 и MAP2 описаны как переменные типа COUNTRIES, то правомерны такие операторы присваивания:

```
MAP1 := [FRANCE]
MAP1 := MAP1 + [GERMANY]
```

```
MAP2 := MAP1
MAP1 := MAP1 * (MAP2 + [ITALY])
MAP2 := MAP2 - [SPAIN] - MAP1
```

Пример 14.2 демонстрирует программу, которая для образования простых чисел использует множества. Алгоритм, используемый в этой программе, известен под названием *решето Эратосфена.* Начиная с набора, состоящего

---

**Пример 14.2. Решето Эратосфена**

```
program SIEVE(INPUT, OUTPUT);
const
    MAXPRIME = 15;
var
    PRIMES : set of 2 .. MAXPRIME;
    COUNT, MULTIPLE : INTEGER;
begin
    WRITELN('Простыми числами меньшими ', MAX-
                                        PRIME, 'являются: ');
    PRIMES := [2 .. MAXPRIME];
    for COUNT := 2 to MAXPRIME do
        if COUNT in PRIMES then
            begin
                WRITELN(COUNT);
                for MULTIPLE := 1 to (MAXPRIME div
                    COUNT) do
                    PRIMES := PRIMES - [COUNT *
                    MULTIPLE]
            end
end.
```

---

из всех целых чисел в интервале 2...MAXPRIME, программа при помощи цикла *for* проверяет каждое целое число, входящее в множество. Если целое число является элементом множества, то оно печатается и из множества удаляются все целые числа, кратные данному числу.

Программа печатает все простые числа в интервале 2..MAXPRIME. (Конечно, не представляет труда модифицировать программу таким образом, чтобы верхняя граница множества была входной переменной, а не символической константой.) По ходу работы программы числа, не являющиеся простыми, удаляются из множества PRIMES:

| *Вывод* | *PRIMES* |
|---|---|
| | [2.. 15] |
| 2 | [3, 5, 7, 9, 11, 13, 15] |
| 3 | [5, 7, 11, 13] |
| 5 | [7, 11, 13] |
| 7 | [11, 13] |
| 11 | [13] |
| 13 | [ ] |

Следует обратить внимание на терм [COUNT * MULTIPLE], который находится во вложенном цикле *for.* Является ли это выражение операцией пересечения? Нет, это обычное умножение. Использование арифметических выражений для получения элементов множества является совершенно правомерным действием. Нельзя допустить другого толкования операции, так как тип компонент множества не может быть множеством.

### 14.3. Сравнение множеств

Ранее уже рассматривался оператор *in,* который позволяет проводить проверку на принадлежность множеству. Хотя проверка при помощи оператора *in* является очень полезной операцией, в ряде случаев она оказывается весьма громоздкой, если вообще возможной. Рассмотрим переменные MAP1 и MAP2, представляющие множества, из разд. 14.2. Для определения равенства этих множеств (т. е. совпадения элементов, составляющих эти множества) необходимо написать нечто подобное:

```
EQUAL := TRUE;
for MEMBER := ENGLAND to ITALY do
      if (MEMBER in MAP1) and not (MEMBER in
      MAP2) then
          EQUAL := FALSE
      else
          if not (MEMBER in MAP1) and (MEMBER in
            MAP2) then
            EQUAL := FALSE;
```

Можно немного улучшить написанное, но не избежать всех затруднений до конца:

```
EQUAL := TRUE;
for  MEMBER := ENGLAND to ITALY do
     if (MEMBER in MAP1) <> (MEMBER in MAP2) then
          EQUAL := FALSE;
```

К счастью, в такого рода циклах нет необходимости. Вместо этого в языке Паскаль используются булевы операторы отношений над множествами. Проверка равенства двух множеств записывается следующим булевым выражением:

```
MAP1 = MAP2
```

В качестве оператора отношений между множествами можно использовать символ неравенства:

```
MAP1 <> MAP2
MAP1 - MAP2 <> [FRANCE]
MAP1 + MAP2 <> [ENGLAND..ITALY]
```

Булевы операторы <= и >= над множествами используются для того, чтобы показать, что одно множество является подмножеством другого множества (или наоборот). Одно множество *меньше, чем* другое множество, если оно является подмножеством второго множества, и одно множество *больше, чем* другое множество, при обратном отношении. Следующие булевы выражения имеют значение true:

```
[ENGLAND, FRANCE, ITALY] >= [ENGLAND, FRANCE]
[ENGLAND, FRANCE] >= [ENGLAND, FRANCE]
[ENGLAND, FRANCE] >= [ENGLAND]
[ENGLAND..ITALY] >= [ENGLAND, GERMANY, SPAIN, ITALY]
[SPAIN] <= [FRANCE, SPAIN]
[ENGLAND, ITALY] <= [ENGLAND, GERMANY, ITALY]
```

Следующие булевы выражения имеют значение FALSE:

```
[FRANCE, ITALY] >= [ENGLAND]
[FRANCE, ITALY] >= [FRANCE, SPAIN]
[FRANCE, ITALY] <= [ITALY]
[FRANCE, ITALY] <= [GERMANY, ITALY]
[ITALY..ENGLAND] >= [ENGLAND]
```

Последнее выражение нуждается в пояснениях. Так как ITALY больше, чем ENGLAND, то в действительности терм [ITALY..ENGLAND] представляет пустое множество. Это правило выполняется для всех множеств, описываемых при помощи интервала. Например, множество [10..1] также трактуется как пустое множество.

ВОПРОСЫ

*1. Какое из следующих описаний неверно?

а)
```
type
    MATRIX = ARRAY [1..10] of BOOLE
    ONESET = set of MATRIX
```

б)
```
type
    EUROPE = (ENGLAND, FRANCE, GE
    NY, SPAIN, ITALY);
    ONESET = set of EUROPE;
```

в)
```
type
    DIGITS = '0'..'9';
    ONESET = set of DIGITS;
    MATRIX = packed array [DIGITS] of ONESET;
```

г)
```
type
    ONESET= set of REAL;
```

д)
```
type
    ONESET = set of BOOLEAN;
```

*2. Какие из следующих булевых выражений имеют значения true?

а) 2 in [0..10]
б) [–0...3] >= [0]
в) [0..5] * [3..7] <> [3, 4, 5]
г) [RED, GREEN, BLUE] <= [RED, GREEN, BLUE]
д) [5] + [2] = [7]
е) [5 + 2] = [7]

3. Для группы людей имеется информация о их занятиях во время досуга. Эта информация задается в форме четырех множеств: SKIERS, SWIMMERS, GOLFERS и BOWLERS. Идентификатор человека присутствует в том или ином множестве, если он или она занимаются этим видом спорта в свободное время. Конькобежцы указаны в множестве SKIERS, пловцы — в SWIMMERS, игроки в гольф — в GOLFERS, игроки в кегли — в BOWLERS. Напишите выражения, которые позволяют получить списки людей, которые:

а) пловцы или конькобежцы;
б) игроки в гольф и кегли;
в) пловцы, но не игроки в кегли;
г) конькобежцы или игроки в гольф, но не конькобежцы и игроки в кегли;

д) пловцы и игроки в гольф, но не игроки в кегли или конькобежцы.

4. Сделайте описание функции VOWELCOUNT, которая в качестве аргумента получает литерную строку; результатом выполнения функции будет число гласных в этой строке. Например, VOWELCOUNT ('Привет') равно 2.

# Глава 15
# Записи

## 15.1. Переменные типа RECORD

Довольно часто вполне оправданным является представление некоторых элементов в качестве составных частей другой, более крупной логической единицы. Представляется естественным сгруппировать информацию о номере дома, названии улицы и городе в единое целое и назвать адресом, а объединенную информацию о дне, месяце и годе рождения — датой. В языке Паскаль для представления совокупности разнородных данных служит структурированный тип *запись*.

Запись и массив схожи в том, что обе эти структуры составлены из ряда отдельных компонент. В то же время, если компоненты массива должны быть одного типа, записи могут содержать компоненты разных типов. Приведем описание переменной, имеющей структуру записи:

```
var
    ADDRESS : record
          HOUSENUMBER : REAL;
          STREETNAME : STRING;
          CITYNAME : STRING;
          STATENAME : packed array [1..2] of CHAR;
          ZIPCODE : INTEGER
    end;
```

Каждая компонента записи называется *полем*. В переменной ADDRESS поле с именем HOUSENUMBER само является переменной типа REAL, поле STREETNAME — двадцатисимвольной строкой и т. д. Для того чтобы обратиться к некоторому полю записи, следует написать имя переменной и имя поля. Эти два идентификатора должны разделяться точкой. Оператор, который присваивает полю HOUSENUMBER значение 2300, выглядит так:

```
ADDRESS.HOUSENUMBER := 2300
```

Таким же образом присваиваются значения другим полям записи ADDRESS:

```
ADDRESS.STREETNAME := 'Центральная улица';
ADDRESS.CITYNAME := 'Город';
ADDRESS.STATENAME := 'NY';
ADDRESS.ZIPCODE := 12345;
```

Отметим, что поля STREETNAME и CITYNAME имеют одинаковый тип. Поскольку в описании эти поля могут располагаться в любом порядке, то можно сократить описание записи с полями одинакового типа. Сокращенное описание записи ADDRESS выглядит следующим образом:

```
var
    ADDRESS : record
                  HOUSENUMBER : REAL;
                  STREETNAME,CITYNAME :
                  STRING;
                  STATENAME : packed array [1..2] of
                  CHAR;
                  ZIPCODE : INTEGER
              end;
```

Каждое поле записи ADDRESS можно рассматривать как обычную переменную, которую можно напечатать или использовать в расчетах. Вместе с тем запись может рассматриваться как единое целое. Предположим, имеется следующее описание:

```
type
    DATE = record
               DAY : 1..31;
               MONTH : (JAN, FEB, MAR, APR, MAY,
                        JUN, JUL, AUG, SEP, OCT,
                        NOV,DEC);
               YEAR: INTEGER
        end;
var
          HISBIRTH, MYBIRTH : DATE
```

После приведенного описания переменные HISBIRTH и MYBIRTH имеют типа DATE. В свою очередь YEAR является переменной типа INTEGER. Помимо действий над отдельными полями записей HISBIRTH и MYBIRTH можно выполнять операции над всей записью. Следующий

оператор присваивания устанавливает равенство значений записей HISBIRTH и MYBIRTH:

```
HISBIRTH := MYBIRTH;
```

Это присваивание эквивалентно следующей последовательности операторов:

```
HISBIRTH.DAY := MYBIRTH.DAY;
HISBIRTH.MONTH := MYBIRTH.MONTH;
HISBIRTH.YEAR := MYBIRTH.YEAR;
```

Для переменных одного типа можно проверить выполнение отношения равенства или неравенства. Как и в случае массивов, допустимы лишь операторы '=' и '<>'. После выполнения приведенных выше присваиваний следующее булево выражение будет иметь значение TRUE :

```
MYBIRTH = HISBIRTH
```

В примере 15.1 описана процедура, которая получает запись в качестве параметра-значения. Дату *3 сентября 1961* эта процедура печатает в сокращенной форме *9-3-1961.* (Аналогично массивам записи могут использоваться либо в качестве параметров-значений, либо в качестве параметров-переменных.)

---

**Пример 15.1. Распечатка дат**

```
prodecure WRITEDATE(ONEDATE : DATE);
(*Эта процедура печатает даты в стандартной форме
  месяц-день-год, используя формат (MM-DD-YYYY).*)
begin
        WRITE(ORD(ONEDATE.MONTH) + 1);
        WRITE('-');
        WRITE(ONEDATE.DAY : 2);
        WRITE('-');
        WRITE(ONEDATE.YEAR : 4)
end; (* процедуры WRITEDATE *)
```

---

Так как на тип компонент массива не накладывается ограничений, то можно образовать массив, компонентами которого являются записи. Приведем описание такого массива:

```
BIRTHDAYS : array[1..PERSONS] of DATES
```

Принимая во внимание предыдущее описание DATES, можно заключить, что компоненты BIRTHDAYS являются записями. Ниже приводится возможный вариант распечатки массива BIRTHDAYS:

```
for COUNT := 1 to PERSONS do
      begin
            WRITEDATE(BIRTHDAYS[COUNT]);
            WRITELN('  ')
      end;
```

Чтобы обратиться к некоторому полю определенной записи массива, следует определить имя массива, индекс интересующей записи и имя необходимого поля. Следующий оператор печатает содержимое поля YEAR записи BIRTHDAYS[3] :

```
WRITELN(BIRTHDAYS[3].YEAR)
```

Как и в случае массивов, на типы компонент записи не накладывается каких-либо ограничений. Поля могут быть массивами, множествами или даже записями. Последний случай будет рассмотрен в ряде примеров последующих разделов.

## 15.2. Оператор WITH

Приведем описание массива, компоненты которого являются записями:

```
var
   PAYROLL : array [1..WORKERS] of
   record
  FIRSTNANE, LASTNAME : STRING;
  RESIDENCE : record
                    HOUSENUMBER: REAL;
                    STREETNAME : STRING;
                    CITYNAME : STRING;
                    STATENAME : packed array [1..2]
                    of CHAR;
                    ZIPCODE : INTEGER
                end;
   PHONE : record
              AREACODE, EXCHANGE : 1..999;
              LINE : 1..9999
           end;
   PAYSCALE :'A' .. 'G'
   end;
```

Отметим, что два поля RESIDENCE и PHONE являются записями. Как выполнить обращение к полям этих записей? Как распечатать условный шифр рабочего № 7? Поскольку это поле располагается во вложенной записи, то следует указать как имя самой записи, так и имя записи, в которую данная запись входит:

```
WRITELN(PAYROLL[7].RESIDENCE.ZIPCODE)
```

Аналогично приведенное присваивание корректирует шифр участка рабочего № 23:

```
PAYROLL[23].PHONE.AREACODE := 804
```

Оператор *if*, представленный ниже, выполняет проверку инициала рабочего № 58:

```
if PAYROLL[58].LASTNAME[1] in ['T' .. 'Z'] then...
```

Соблюдение всех правил перечисления индексов и имен полей при составлении ссылок является довольно утомительным занятием, часто приводящим к ошибкам. В некоторых программах, содержащих большое число обращений к одному и тому же полю, такое положение приводит к однообразному повторению. Чтобы облегчить выполнение многократных ссылок для описанных структур, в языке Паскаль вводится оператор *with*.

---

WITH

Общая форма записи:

with *имя переменной* do оператор

В рамках оператора, определяемого внутри оператора with (или составного оператора), к полям определяемой переменной можно обращаться просто по имени. Поэтому префиксные идентификаторы могут опускаться.

**Примеры:**

```
with PAYROLL[7].RESIDENCE do
     ZIPCODE := 23186
for EMPLOYEE := 1 to WORKERS do
     with PAYROLL[EMPLOYEE] do
          if PAYSCALE < 'G' then
          PAYSCALE := SUCC(PAYSCALE)
```

---

Оператор *with* позволяет более компактно представлять часто используемые переменные. Покажем это на примере фрагмента программы, печатающего адрес рабочего № 14:

```
with PAYROLL[14].RESIDENCE do
    begin
        WRITELN(HOUSENUMBER,  STREETNAME)
        WRITELN(CITYNAME, ',   ', STATENAME,
        ZIPCODE)
    end;
```

В рамках составного оператора, следующего за *with,* каждое обращение к имени поля автоматически связывается с записью PAYROLL[14].RESIDENCE. Печать адресов всех рабочих выполняется при помощи следующего оператора цикла:

```
for EMPLOYEE := 1 to WORKERS do
      with PAYROLL[EMPLOYEE].RESIDENCE do
          begin
              WRITELN(HOUSENUMBER,
              STREETNAME);
              WRITELN(CITYNAME, ', ',
              STATENAME, ZIPCODE)
          end;
```

Операторы *with* могут быть вложенными. Приведенные три оператора эквивалентны друг другу:

```
PAYROLL[EMPLOYEE].RESIDENCE.HOUSENUMBER
                                    := 55
with PAYROLL[EMPLOYEE].RESIDENCE do
    HOUSENUMBER := 55
with PAYROLL[EMPLOYEE] do
    with RESIDENCE do
        HOUSENUMBER := 55
```

Однако недопустимым является использование вложенных операторов *with,* в которых указываются поля одного типа, поскольку возникает неоднозначность конструкции. По этой причине приведенное использование вложенных операторов *with* является неверным:

```
with PAYROLL[5] do
    with PAYROLL[17] do
        PAYSCALE := 'A'
```

Следует очень внимательно подходить к использованию вложенных операторов with, применение которых не только может привести к ошибкам (как в приведенном выше примере), но также к потере наглядности структуры

программы. Хотя оператор *with* является стандартным средством сокращения, его полезность должна еще и проявиться. Конечной целью всякого хорошего программиста является написание понятной, а не короткой программы.

### 15.3. Записи с вариантами

Записи, описанные в разд. 15.1 и 15.2, — это *записи с фиксированными частями.* Они имеют в различных ситуациях строго определенную структуру. Соответственно *записи с вариантами* в различных ситуациях могут иметь различную структуру.

Предположим, что написана программа для введения списка библиографических ссылок. Если известно, что все входы в этом списке — ссылки на книги, то можно использовать следующее описание:

```
const
      MAXREFS = ...
type
      ENTRY = record
                  AUTHOR, TITLE, PUBLISHER, CITY :
                                                STRING;
                  YEAR : 1..9999
                  end;
var
      REFLIST : array[1..MAXREFS] of ENTRY;
```

Что́ произойдет, если некоторые из входов не являются ссылками на книги, а содержат ссылки на журнальные статьи? Если ограничиваться только записями с фиксированными частями, то следует описать различные массивы для каждого вида записей. Использование записей с вариантами позволяет образовать структуру, каждый вход которой соответствует содержанию записи. Опишем новый тип, в котором перечислены различные входы:

```
ENTRYTYPE = (BOOK, MAGAZINE);
```

Теперь можно привести скорректированное описание ENTRY:

```
ENTRY = record
            AUTHOR, TITLE : STRING;
            YEAR : 1..9999;
            case ENTRYTYPE of
                  BOOK :(PUBLISHER, CITY :
                              : STRING);
                  MAGAZINE : (MAGNAME :
                                    : STRING; VOLUME,
                                    ISSUE : INTEGER)
        end;
```

Это описание делится на две части: *фиксированную* и *вариантную.* Поля AUTHOR, TITLE и YEAR составляют фиксированную часть. Оставшаяся часть описания ENTRY образует вариантную часть, структура которой, подобно хамелеону, может меняться в пределах двух альтернативных определений.

Первая строка вариантной части представляет оператор *case,* который отличается тем, что в качестве селектора применяется идентификатор типа. Значения ENTRYTYPE используются в качестве имен двух альтернатив определения записи. Когда эта компонента REFLIST имеет значение BOOK, можно обращаться к следующим полям:

AUTHOR
TITLE
YEAR
PUBLISHER
CITY

С другой стороны, когда она принимает значение MAGAZINE, то можно обращаться к таким полям:

AUTHOR
TITLE
YEAR
MAGNAME
VOLUME
ISSUE

В такой ситуации может возникнуть естественный вопрос: как программа может хранить информацию о текущем состоянии каждой записи? Другими словами, каким образом можно узнать, что REFLIST[3] содержит ссылку на книгу, а REFLIST[4] — ссылку на журнал?

Естественное решение этой проблемы заключается в добавлении в каждой записи нового поля, называемого *полем тега.* Язык Паскаль позволяет за счет совмещения задать описание поля тега в сокращенной форме:

```
ENTRY = record
          AUTHOR, TITLE: STRING;
          YEAR : 1..9999;
          case TAG : ENTRYTYPE  of
          BOOK : (PUBLISHER, CITY : STRING);
          MAGAZINE : (MAGNAME : STRING;
                      VOLUME,  ISSUE : INTEGER)
        end;
```

Поле, названное TAG, является переменной типа ENTRYTYPE. Когда запись содержит ссылку на книгу, TAG следует присвоить значение BOOK. Когда запись содержит ссылку на журнал, TAG следует присвоить значение MAGAZINE. В приведенной последовательности операторов в REFLIST[12] помещается ссылка на книгу:

```
REFLIST[12].TAG := BOOK;
REFLIST[12].AUTHOR := 'Thomas Hobbes';
REFLIST[12].TITLE := 'Leviathan';
REFLIST[12].YEAR := 1651;
REFLIST[12].PUBLISHER := 'Andrew Crooke';
REFLIST[12].CITY := 'London';
```

Для определения состояния записи с вариантами достаточно проверить значение поля тега. В примере 15.2 приведена процедура, которая получает значение, относящееся к типу ENTRY, и распечатывает его.

Что́ произойдет, если программа попытается обратиться к полю, которое отсутствует в текущем состоянии записи? В языке Паскаль отсутствует контроль за возникновением подобного рода ошибочных ситуаций[1]. Следовательно, операция будет выполнена, но приведет к бессмысленному

[1] Это справедливо для большинства известных реализаций. Однако в стандарте языка Паскаль оговорено, что если осуществляется ссылка на поля варианта, отличные от полей текущего варианта, то возникает состояние ошибки. Хочется надеяться, что и эта проверка на этапе выполнения будет осуществляться в новых реализациях языка Паскаль. — *Прим. ред.*

**Пример 15.2. Распечатка вариантной записи**

```
procedure PRINTREF(CITATION : ENTRY);
begin
      WRITELN(CITATION.AUTHOR);
      WRITELN(CITATION.TITLE);
      WRITELN(CITATION.YEAR);
      if CITATION.TAG = BOOK then
          WRITELN(CITATION.PUBLISHER,',',
                                        CITATION.CITY)
   else
          begin
                WRITELN(CITATION.MAGNAME);
                WRITELN(CITATION.VOLUME, '-',
                                        CITATION.ISSUE)
          end
end; (* процедуры PRINTREF *)
```

результату. Если REFLIST[12] содержит ссылку на книгу, то выполнение приведенного оператора приведет к непредсказуемому результату:

```
WRITELN(REFLIST[12].VOLUME)
```

Вариантная часть может содержать произвольное число альтернатив. Хотя перечисляемые типы предпочтительней, так как они более понятны, тем не менее для именования альтернатив записи с вариантами могут использоваться идентификаторы произвольного порядкового типа. Ниже приводится описание записи, в котором для этих целей используется интервал типа INTEGER:

```
type
    ITEMKIND = 1..3;
    STOCKITEM = record;
                      PARTNUMBER : INTEGER;
                      case KIND : ITEMKIND of
                         1 : (LITERS : REAL);
                         2 : (GALLONS : REAL);
                         3 : (FLUIDOUNCES : REAL)
                  end;
```

Очевидно, что один и тот же идентификатор поля не может дважды использоваться при описании записи, даже если он применяется в определении различных альтернатив записи с вариантами. Если же это условие

не выполняется, то обращение к такому идентификатору приведет к непредсказуемому результату. Отметим также, что описание записи с вариантами может иметь единственный закрывающий оператор *end.* Поскольку любая запись может иметь лишь одну вариантную часть, то *end,* который является индикатором конца описания записи, служит для обозначения конца и ее вариантной части.

## 15.4. Упакованные записи

Упакованные записи, подобно упакованным массивам, позволяют программисту за счет уменьшения скорости обработки получить более экономное использование пространства оперативной памяти. В большинстве программ указанный компромисс не является первостепенным вопросом, однако следует знать о такой возможности. Ниже приведено описание переменной, представляющей упакованную запись:

```
ADDRESS : packed record
              HOUSE_NUMBER : REAL;
              STREET_NAME,  CITY_NAME : STRING
              [20];
              STATE_NAME : STRING[2];
              ZIP_CODE : INTEGER
            end;
```

Отношение между упакованными и неупакованными записями точно такое же, как и между упакованными и неупакованными массивами. Ограничения, накладываемые на упакованные массивы, применимы и к упакованным записям. Во-первых, упакованные записи не могут проверяться при помощи булевых операторов равенства и неравенства. Во-вторых, некоторое поле упакованной записи не может передаваться в качестве параметра-переменной.

ВОПРОСЫ

*1. Рассмотрите следующие описания:

```
type
    DATE = record
               DAY : 1..31;
               MONTH : 1..12;
               YEAR : 1..9999
           end;
    REMINDER = record
                 MESSAGE : array [1..5] of STRING;
                 EVENT : DATE
               end;
var
    TODAY : DATE;
    MEMOS : array [1..100] of REMINDER;
    CALENDAR : array [1..365] of DATE;
```

Какой тип, если он определен, у следующих идентификаторов?

а) TODAY.YEAR
б) MEMOS[2]
в) MEMOS[4].MONTH
г) CALENDAR[200]
д) MEMOS[16]MESSAGE[2]
е) MEMOS[16].MESSAGE[2][1]
ж) CALENDAR[1].DATE
з) MEMOS[10].EVENT

2. Опишите под именем FIGURE вариантную запись. Если переменная типа FIGURE представляет круг, то она должна содержать радиус соответствующей окружности. Если эта переменная представляет прямоугольник, то она должна содержать длину сторон. Если эта переменная представляет треугольник, то она должна содержать величину угла и длины двух сторон, образующих этот угол.

а) Напишите процедуру, которая запрашивает и получает значение типа FIGURE от пользователя.
б) Напишите функцию, которая получает на входе значение типа FIGURE и вычисляет площадь фигуры.
в) Напишите функцию, которая получает на входе значение типа FIGURE и вычисляет периметр фигуры.
г) Напишите булеву функцию, которая на входе получает

два значения типа FIGURE и определяет, помещается ли первая фигура внутри второй.

*3. Опишите запись, которая представляет собой некоторый отчетный документ по собираемой информации. По каждому покупателю эта информация включает номера квитанций, задолженность по каждой квитанции. (Предполагается, что максимальное число квитанций по каждому покупателю равно десяти.) Напишите программу, которая должна указать покупателя с самой старой неоплаченной квитанцией.

4. Выполните описание переменной типа запись, которая ведет учет получаемой информации. По каждому из потребителей данная информация должна включать число накладных, неоплаченный баланс и дату по отдельной накладной. (Предполагается, что каждый из потребителей имеет максимум десять накладных.) Далее напишите программу, которая принимает от пользователя получаемую информацию и затем указывает потребителя с самой старой неоплаченной накладной.

# Глава 16
# Текстовые файлы

## 16.1. Описание текстовых файлов

Различными комбинациями структурированных типов языка Паскаль можно создать огромное множество структур данных. Однако при этом все же существует два ограничения. Во-первых, нельзя создать программы, которые формируют значения и сохраняют их для последующего использования другими программами. Однако необходимость механизма длительного хранения информации является очевидной. Фактически в ряде приложений без него нельзя обойтись. Если требуется написать программы для ведения платежных ведомостей или счетов, то нельзя требовать, чтобы при каждом использовании программ все элементы информации заново вводились вручную. Для решения этой проблемы в языке Паскаль предусмотрен еще один структурированный тип, называемый *файлом,* который позволяет сохранить значения и при необходимости восстановить их.

Во-вторых, нельзя разработать программы, которые взаимодействуют с устройствами ввода-вывода, отличными от терминала пользователя. Однако вычислительные системы имеют в своем составе много вспомогательных внешних устройств, таких, как устройства печати, аналого-цифровые преобразователи и т. д. Безусловно, было бы странным иметь эти устройства и не использовать их. Файлы позволяют снять и это ограничение.

Файловая переменная выступает в роли «окна» между программой и внешним устройством. Поэтому в программе на языке Паскаль каждая файловая переменная логически связана с некоторым внешним устройством. В большинстве случаев файлы для хранения значений логически связаны с дисковыми накопителями, использующими магнитный носитель для хранения и воспроизведения большого объема данных. Например, если файл, названный MYFILE, логически связан с конкретным дисковым накопителем, то и все данные, помещаемые в MYFILE, будут храниться на этом дисковом накопителе.

*Текстовый файл* — это файл, состоящий из литерных строк. Может показаться неожиданным, что во всех ранее приведенных программах были использованы текстовые файлы. В частности, до сих пор использовались два стандартных файла с именами INPUT и OUTPUT. Используемые процедуры READ и READLN считывают значения литерных строк из файла INPUT, а процедуры WRITE и WRITELN помещают литерные строки в файл OUTPUT. Безусловно, предполагается, что в качестве внешнего устройства для файлов INPUT и OUTPUT используется терминал пользователя.

Прежде чем использовать текстовые файлы, отличные от INPUT или OUTPUT, их следует описать. В языке Паскаль предусмотрены два стандартных типа TEXT и INTERACTIVE. Последний тип соответствует файлам, которые логически связаны с терминалами, в то время как первый тип соответствует файлам, которые логически связаны с другими устройствами. Файлы INPUT и OUTPUT имеют тип INTERACTIVE. Следовательно, любая программа на языке Паскаль содержит следующее, невидимое программистам описание:

```
INPUT, OUTPUT : INTERACTIVE
```

Следует отметить, что файлы не могут быть включены в другие структурированные типы. Например, массив не может иметь тип компонент TEXT, поля записей также не могут быть файлами[1].

## 16.2 Открытие текстовых файлов

Следующим действием в работе с текстовыми файлами является «открытие» файла. Для открытия файлов в языке Паскаль предусмотрены две процедуры RESET и REWRITE. Действие этих процедур зависит от типа устройства, к которому логически подсоединен этот файл. Внешние устройства делятся на устройства с *блочной структурой* и устройства *с символьной структурой.*

К устройствам с блочной структурой относятся устройства, которые постоянно хранят данные в форме дискрет-

[1] В наиболее общем виде это правило звучит так: «Никакая компонента никакого структурированного типа не может быть файлом». Можно, конечно, обсуждать теоретические концепции, например, массива файлов или файла файлов. На практике, однако, это просто неприемлемо. — *Прим. ред.*

ных блоков. (Дисковые накопители являются примером устройства с блочной структурой.) Последовательность данных, расположенных на устройстве с блочной структурой, называется *внешним файлом.* (Такое название позволяет отличать это понятие от файловых переменных, используемых в программе.) Так как устройство с блочной структурой может одновременно хранить несколько внешних файлов, то каждый файл на устройстве должен иметь уникальное имя. Для того чтобы установить логическую связь между файловой переменной и устройством с блочной структурой, следует показать соответствие между именем устройства и именем внешнего файла. Этого можно добиться, записав оба имени вместе и разделив их двоеточием. Например, чтобы определить, что внешний файл, называемый 'ESSAY.TEXT', располагается на устройстве с именем '№ 5', следует написать

'№ 5 : ESSAY.TEXT'

---

**RESET и REWRITE**

Общая форма записи:

RESET (*файловая переменная*)
RESET (*файловая переменная, внешнее имя)*
REWRITE (*файловая переменная*)
REWRITE (*файловая переменная, внешнее имя*)

*Внешнее имя* — это строковое выражение, определяющее имя устройства, с которым будет логически связана *файловая переменная.* Если *внешнее имя* опущено, *файловая переменная* трактуется как временный файл, для которого ЭВМ назначает внешнее устройство на период работы программы

**Примеры:**

```
RESET (SCRATCHFILE)
RESET (OLD _ FILE, 'ADDRESS.TEXT')
REWRITE (LEDGER)
REWRITE (NEW _ FILE, 'REMOUT :')
RESET (GAME, CONCAT (USER _ NAME, '.TEXT'))
```

---

При использовании процедуры RESET для устройства с блочной структурой ЭВМ просматривает содержимое устройства, стремясь найти указанный внешний файл. Если внешний файл найден, то устанавливается логи-

ческая связь между файловой переменной и внешним файлом. В противном случае вырабатывается условие ошибки.

При использовании процедуры REWRITE для устройства с блочной структурой ЭВМ создает новый внешний файл с указанным именем. Первоначально вновь созданный внешний файл является пустым. Если внешний файл с таким именем уже существует, то никаких действий не выполняется. Работа процедуры оканчивается установкой логической связи между файловой переменной и созданным внешним файлом.

• К устройствам с символьной структурой относятся устройства, которые осуществляют посимвольную передачу данных. Хотя данные, посланные на устройство с символьной структурой, могут в какой-то форме запоминаться, впоследствии их нельзя воспроизвести в форме внешнего файла. По этой причине логическая связь между файловой переменной и устройством с символьной структурой устанавливается путем определения имени устройства. На ЭВМ, которые имеют данную реализацию языка Паскаль, типовыми являются следующие имена устройств:

| | |
|---|---|
| 'CONSOLE:' | для обозначения терминала пользователя; |
| 'PRINTER:' | для обозначения устройства печати, если оно имеется; |
| 'REMIN:' | для обозначения вспомогательной линии приема, если таковая имеется; |
| 'REMOUT' | для обозначения вспомогательной линии передачи, если таковая имеется. |

При использовании процедуры RESET для устройства с символьной структурой ЭВМ подготавливает устройства к выполнению операции чтения. Если устройство не может выполнить операцию чтения, как в случае PRINTER : или REMOUT : , то вырабатывается условие ошибки. Приведенный ниже оператор открывает файловую переменную, названную MODEM, и устанавливает логическую связь между этой переменной и вспомогательной линией передачи:

```
RESET(MODEM,'REMIN :')
```

При использовании процедуры REWRITE для устройства с символьной структурой ЭВМ подготавливает устройство к выполнению операции записи. Будет выработано условие ошибки, если устройство не приспособлено к

выполнению операции записи, как в случае REMIN :. Если строковая переменная DEVICE _ NAME равна 'PRINTER :', то приведенный оператор будет открывать файловую переменную LISTING и устанавливать логическую связь с устройством печати:

```
REWRITE(LISTING, DEVICE _ NAME)
```

Если устройство печати не подсоединено к системе, то этот оператор приведет к возникновению условия ошибки. Используя стандартную функцию IORESULT, можно получить результат работы процедур RESET или REWRITE. Эта функция имеет значение, равное нулю, если последняя операция открытия файла выполнена успешно, и некоторое положительное число — в противном случае. Чтобы использовать функцию IORESULT, нужно предварительно указать ЭВМ, что не следует прерывать работу программы при неудачной попытке выполнить операцию. Это достигается тем, что процедуры RESET и REWRITE обрамляются комментариями (*$I−*) и (*$I+*). Эти комментарии называются *переключателями.* Первый комментарий блокирует контроль ошибок при работе с файлом, второй — разблокирует его. Ниже приведена последовательность операторов, которые применяются при использовании IORESULT :

```
(*$I − *)
REWRITE (LISTING, DEVICE _ NAME);
(*$I + *)
if IORESULT <> 0 then
   WRITELN (DEVICE _ NAME,' − недоступно.');
```

Если используется функция IORESULT, то ее результат должен проверяться непосредственно после операции над файлом. Если между проверяемой операцией и функцией IORESULT произойдет другая операция ввода-вывода (например, WRITELN), то IORESULT будет отражать результат выполнения самой последней операции. Можно сохранить результат выполнения процедур RESET или REWRITE в булевой переменной, обеспечив возможность дальнейшего использования полученного значения.

### 16.3. Операции над текстовыми файлами

После того как текстовый файл открыт, становится возможным выполнение ряда процедур, применяемых в

языке Паскаль для работы с текстовыми файлами: READ, READLN, WRITE и WRITELN. По умолчанию, операции READ и READLN выполняют по отношению к файлу INPUT, а WRITE и WRITELN — по отношению к файлу OUTPUT. Чтобы получить доступ к другому файлу, следует указать имя файловой переменной первым элементом списка параметров. Приведенный оператор выводит литерную строку в файловую переменную с именем BACK _ ORDERS :

```
WRITELN(BACK _ ORDERS, 'Номер раздела', PART)
```

Поскольку, по умолчанию, WRITELN предполагает файл OUTPUT, то следующие операторы являются эквивалентными:

```
WRITELN ('Это литерная строка.')
WRITELN (OUTPUT, 'Это литерная строка.')
```

В примере 16.1 показана программа, которая передает данные из устройства с блочной структурой в устройство

---

**Пример 16.1. Распечатка внешнего файла**

```
program PRINT _ FILE;
(* Эта программа распечатывает содержимое внешнего
   файла на устройстве PRINTER :*)
var
    IN _ FILE, OUT _ FILE : TEXT;
    IN _ FILE _ NAME : STRING;
    LINE _ OF _ DATA : STRING;
begin
    WRITELN ('Введите, пожалуйста, имя внешнего
    файла,');
    WRITELN ('содержимое которого должно быть распе-
    чатано.');
    READLN (IN _ FILE _ NAME);
    RESET (IN _ FILE, IN _ FILE _ NAME);
    REWRITE (OUT _ FILE, 'PRINTER :');
    while not EOF (IN _ FILE) do
        begin
            READLN(IN _ FILE, LINE _ OF _ DATA);
            WRITELN (OUT _ FILE,    LINE _ OF _
            DATA)
        end
end.
```

---

с символьной структурой. Функция EOF используется для формирования условия конца внешнего файла. До тех пор пока в IN – FILE содержатся несчитанные литеры, функция EOF(IN – FILE) имеет значение FALSE. (Если функция EOF выполняется над файловой переменной, имеющей тип INTERACTIVE, то ее работа немного отличается от той, которая описана в разд. 5.3.)

Язык Паскаль содержит третью стандартную файловую переменную, называемую KEYBOARD. Подобно INPUT, KEYBOARD имеет тип INTERACTIVE и представляет данные, вводимые с терминала пользователя. Различие между этими двумя файловыми переменными заключается в различной реакции на так называемый *эхо-эффект.* Когда данные считываются из файла INPUT, они автоматически отображаются на терминале пользователя (эхо-эффект). С другой стороны, при считывании данных из файла KEYBOARD эхо-эффект отсутствует, т. е. ЭВМ считывает данные с клавиатуры терминала, но не посылает их на устройство отображения терминала. Ниже приведен фрагмент программы, который позволяет ввести пароль, делая его невидимым для других:

```
WRITELN ('Введите, пожалуйста, пароль.');
READLN (KEYBOARD, ENTRY);
if ENTRY <> PASSWORD then
    begin
      WRITELN ('Извините, но пароль неверен.');
      EXIT (program)
    end;
```

Текстовые файлы, которые логически связаны с терминалами, по-разному реагируют на выполнение операторов WRITE и WRITELN. Как отмечалось в разд. 3.1, оператор WRITELN приводит к позиционированию курсора или печатающей головки к началу следующей строки. Однако, когда WRITE и WRITELN используются для других устройств, отличие этих операторов проявляется в несколько другой форме. Так же как текстовые файлы, логически связанные с терминалами, могут подразделяться на физические строки, текстовые файлы для других устройств могут делиться на так называемые *логические строки.*

Логической строкой является литерная строка, конец которой отмечается литерой "end-of-line" (конец строки).

Если в текстовом файле хранится много логических строк, то литера «конец строки» отмечает границы логических строк. WRITELN помещает литеру «конец строки» в конец каждой выводимой строки, а WRITE не делает этого. Таким образом, каждая из трех приведенных ниже последовательностей операторов выводит единственную логическую строку в файл REPORT :

```
WRITELN(REPORT, 'abcdef');
```

```
WRITE(REPORT, 'abc');
WRITELN(REPORT, 'def');
```

```
WRITE(REPORT, 'abc');
WRITE(REPORT, 'def');
WRITELN(REPORT);
```

Отношение между READ и READLN аналогично отношению между WRITE и WRITELN. READ игнорирует границы строк, в то время как READLN использует их для определения того, какие данные выбираются. Если в логической строке представлено больше символов, READLN отбрасывает лишние символы. Предположим, что в текстовом файле с именем RENTALS имеется следующая логическая строка:

```
'27 25 31 29'
 ↑
```

Стрелка представляет собой некоторый маркер, который перемещается по файлу. Позиция маркера указывает литеру, которая должна быть считана. После каждой операции READ маркер перемещается на число литер, которые были прочитаны. После выполнения оператора READ(RENTALS, I, J) строка выглядела бы следующим образом:

```
'27 25 31 29'
      ↑
```

В противоположность этому каждая операция READLN перемещает маркер в начало следующей строки. Поэтому оператор READLN(RENTALS, K) считал бы 31 и затем переместил бы курсор за 29. С другой стороны, выполнение операции READ (RENTALS, K) установило бы маркер на первую литеру числа 29, которую позже можно было бы считать.

Для обнаружения конца строки в текстовых файлах используется булева функция EOLN (конец строки).

Если маркер находится на конце строки, то функция EOLN имеет значение TRUE; в противном случае — FALSE. Значения, имеющие тип STRING, должны считываться при помощи READLN, а не READ. Однако, используя READ и EOLN, можно считать строку посимвольно:

```
while not EOLN (RENTALS) do
      begin
            READ (RENTALS, ONE – CHAR); (*ввести
            литеру*)
            WRITE (ONE – CHAR)          (*напечатать
            ее*)
      end;
```

Функции EOLN и EOF могут использоваться без указания файлов в качестве аргументов. По умолчанию, для функций EOLN и EOF предполагается файл INPUT. Таким образом, приведенные операторы *if* являются эквивалентными:

```
if EOLN then...
if EOLN (INPUT) then...
```

После открытия файла процедуры RESET и REWRITE могут использоваться для дальнейшей работы с файлом. В частности, если файл логически связан с устройством, имеющим блочную структуру, RESET и REWRITE могут применяться для перемещения файлового маркера. RESET устанавливает маркер в начало файла. Процедура REWRITE также устанавливает маркер в начало файла, но при этом стирает все данные, хранящиеся в нем. Приведенная ниже последовательность файловых операций иллюстрирует работу этих процедур:

```
REWRITE (DUMMYFILE, '=5:DIGITS.TEXT')   ''
                                         ↑
WRITELN (DUMMYFILE, 2*6, 3+4,4+5)        '12 7 9'
                                            ↑
RESET (DUMMYFILE)                        '12 7 9'
                                         ↑
READLN (DUMMYFILE, I)                    '12 7 9'
                                            ↑
REWRITE (DUMMYFILE)                      ''
                                         ↑
WRITELN (DUMMYFILE, I, I + 1, I + 2)       '12 13 14'
                                                  ↑
RESET (DUMMYFILE)                        '12 13 14'
                                         ↑
READ (DUMMYFILE, J)                      '12 13 14'
                                            ↑
```

После выполнения операции REWRITE, функции EOF и EOLN принимают значение TRUE. Выполнение операции RESET над непустым файлом приводит к тому, что значение функции EOF становится равным FALSE. В примере 16.2 показана функция, которая в качестве аргумента получает файловую переменную и вычисляет число знаков препинания в указанном файле. Функции PUNCTUATION может быть передан любой открытый текстовый файл. Любая процедура или функция, которая получает на входе файловые переменные, должна принимать их как параметры-переменные, а не как параметры-значения. Используя процедуру или функцию, которая получает на вход файловую переменную, следует заботиться о положении маркера файла. Функция PUNCTUATON оставляет маркер в конце файла, и программист должен помнить, что после работы PUNCTUATION следует выполнить RESET или REWRITE.

---

**Пример 16.2. Подсчет знаков препинания**

```
function PUNCTUATION(var CHARFILE : TEXT) : INTE-
    GER;
var
    SYMBOL_COUNT : INTEGER;
    SYMBOL : CHAR;
begin
    SYMBOL_COUNT := 0;
    RESET (CHARFILE);
    while not EOF (CHARFILE) do
        begin
            READ (CHARFILE, SYMBOL);
            if SYMBOL in ['. ', ', ', '; ', ': ', '! ', '? ',
            ' ' ' ', '-'] then
              SYMBOL - COUNT := SYMBOL -
              COUNT + 1
        end;
    PUNCTUATION := SYMBOL - COUNT
end; (*функции PUNCTUATION*)
```

---

Некоторые устройства печати используют управляющие символы в качестве сигналов для прогона страницы или перемещения к началу следующей. Такая операция называется *форматным перемещением.* Для выполнения формат-

ных перемещений в языке Паскаль предусмотрена встроенная функция PAGE, которая помещает символ форматного перемещения в указанную файловую переменную. (Предполагается, что файл логически связан с устройством печати.) Типичное обращение к функции PAGE можно представить следующим образом:

PAGE(HARD_COPY)

Поскольку INPUT и OUTPUT являются файлами типа INTERACTIVE, то аналогично другим файловым переменным они могут быть переданы в качестве аргументов. Однако эти файлы не могут выступать в качестве аргументов операций RESET или REWRITE. (По этой причине нельзя писать в файл INPUT или считывать данные из файла OUTPUT.)

### 16.4. Закрытие текстовых файлов

Всякая файловая переменная, которая открывается, должна и закрываться. После закрытия файла над ним нельзя выполнять какие-либо действия, не повторив открытия файла. Если файловая переменная описана как глобальная, то при завершении программы выполняется ее автоматическое закрытие. Аналогично файловая переменная, описанная в подпрограмме, автоматически закрывается при завершении подпрограммы.

---

**CLOSE**

Общая форма записи:

CLOSE (*файловая переменная*)

CLOSE (*файловая переменная, режим*)

*Файловая переменная* представляет имя файла, который должен закрываться. Параметр *режим* определяет характер операции закрытия. Возможны следующие режимы операции закрытия: LOCK, CRUNCH, NORMAL и PURGE. По умолчанию, принимается значение NORMAL.

**Примеры:**

CLOSE (MY_FILE)
CLOSE (MY_FILE, NORMAL)
CLOSE (MY_FILE, LOCK)
CLOSE (MY_FILE, CRUNCH)
CLOSE (MY_FILE, PURGE)

---

Возможны четыре режима закрытия файла. Для файлов, логически связанных с устройствами с символьной структурой, все режимы равнозначны. Для файлов, логически связанных с устройствами с блочной структурой, параметр режима определяет состояние файла. Если файловая переменная закрывается автоматически, то закрытие происходит в режиме NORMAL. В этом режиме данные, записанные в результате работы программы, не сохраняются. Если внешний файл уже существовал, то содержимое файла остается таким же, как и до работы программы. (Любой внешний файл, созданный по REWRITE, при закрытии в режиме NORMAL ликвидируется.)

Если требуется закрыть файл заранее или в другом режиме, то следует использовать стандартную процедуру CLOSE. Вызов CLOSE приводит к немедленному закрытию файловой переменной, а вовсе не завершению работы программы или подпрограммы. Можно использовать любой режим закрытия. Однако наиболее полезным, по всей видимости, является режим LOCK. В этом режиме внешний файл, созданный по REWRITE, сохраняется. Если новый внешний файл имеет такое же имя, как у уже существующего на этом устройстве файла, то старая версия удаляется.

---

**Пример 16.3. Кодирование текстового файла**

```
program ENCODE;
var
    OLD_FILE, NEW_FILE : TEXT;
    OLD_NAME, NEW_NAME : STRING;
    LINE : STRING;
    COUNT : INTEGER;
begin
    WRITELN ('Введите, пожалуйста, имя внешнего
    файла,');
    WRITELN ('который должен быть закодирован.');
    READLN (OLD_NAME);
    WRITELN ('Введите, пожалуйста, имя нового внешнего
    файла.');
    READLN (NEW_NAME);
            (*-- открытие файлов --*)
```

```
    RESET (OLD _ FILE, OLD _ NAME);
    REWRITE (NEW _ FILE, NEW _ NAME);
              (* — — закодировать OLD _ FILE и скопиро-
              вать его в NEW _ FILE файл — — *)
    while .not EOF (OLD _ FILE) do
          begin
               READLN (OLD _ FILE, LINE);
               for COUNT : = 1 to LENGTH (LINE) do
                   if ORD (LINE [COUNT]) = 255 then
                      LINE [COUNT] : = CHR (0)
                   else
                      LINE    [COUNT] : = SUCC  (LINE
                      [COUNT]);
               WRITELN (NEW _ FILE, LINE)
          end;
          (*— — следует сохранить NEW _ FILE - -*)
    CLOSE (NEW _ FILE, LOCK)
end.
```

---

Режим CRUNCH идентичен режиму LOCK, если только маркер не располагается где-то в середине файла. При использовании режима CRUNCH все данные за адресным маркером теряются. Во внешнем файле сохраняются только данные, расположенные между началом файла и текущим положением маркера.

Режим PURGE идентичен NORMAL, если только ранее не существовало внешнего файла с таким же именем. В отличие от режима NORMAL, при котором ликвидируется новый внешний файл и остается нетронутым старый, PURGE ликвидирует оба файла. В табл. 16.1 приведены особенности работы каждого из четырех режимов закрытия файловой переменной, которая логически связана с внешним файлом.

*Таблица 16.1.* Режимы закрытия файлов

| | NORMAL | LOCK | CRUNCH | PURGE |
|---|---|---|---|---|
| Сохраняется новая версия | Нет | Да | Да | Нет |
| Сохраняется старая версия (если она существует) | Да | Нет | Нет | » |

Текст программы, приведенный в примере 16.3, представляет довольно простой алгоритм кодирования, позволяющий скрыть содержание текстового файла. Программа ENCODE из заданного внешнего файла формирует новый файл, в котором каждая литера является «смещенным» на одну позицию литерным кодом первоначального файла. (Строка 'My name is Bob' будет преобразована к виду 'Nz!obnf!jt!Cpc'.) Образованный файл закрывается в режиме LOCK.

ВОПРОСЫ

*1. Напишите программу для подсчета числа литер и строк в текстовом файле.

2. Напишите программу, которая считывает содержимое текстового файла и распечатывает его в формате, указанном пользователем. Пользователь должен иметь возможность определить максимальное число литер в строке, максимальное число строк на странице, а также интервал между строками (одинарный или двойной). Если логическая строка файла больше заданной длины печатной строки, то следует произвести усечение строки справа. Страницы должны быть пронумерованы в верхнем правом углу.

3. Модифицируйте программу, описанную во втором пункте, таким образом, чтобы она не усекала длинную строку. Новая программа должна найти пробел, по которому можно поделить строку, и остаток разместить на следующей печатной строке.

*4. Модифицируйте программу, приведенную в примере 16.3, таким образом, чтобы она принимала символы с KEYBOARD и в закодированной форме выводила их на устройство отображения терминала (файл OUTPUT).

5. Напишите программу, которая считывает две строки из текстового файла. Каждая строка состоит из цифровых литер. Программа должна найти сумму чисел, представленных этими строками. (Так как строки могут иметь длину до 255 литер, то их нельзя преобразовывать к значениям типа INTEGER для выполнения сложения.) Содержимое текстового файла должно быть заменено строкой, представляющей найденную сумму.

*6. Напишите процедуру, которая на входе получает строку, содержащую имя внешнего файла, и выполняет уничтожение указанного внешнего файла.

# Глава 17
# Динамические структуры данных

## 17.1. Стеки и очереди

*Статические структуры данных* — это такие структуры, в которых всегда содержится постоянное число компонент. Одним из примеров статических структур является массив, другим — запись постоянной длины. С другой стороны, *динамическая структура данных* во время выполнения программы может увеличиваться или уменьшаться. Мы уже рассматривали два вида динамических структур данных: запись с вариантами и множество. Используя в качестве основы встроенные типы данных, в языке Паскаль можно построить другие динамические структуры. Простейшими среди них являются *стек* и *очередь*.

Стек — это линейный список с одной точкой доступа, называемой *вершиной*. Доступ может быть осуществлен только к элементу, расположенному в вершине стека. Новый элемент добавляется в вершину стека; элемент удаляется также из вершины стека. Стек часто называют структурой LIFO [сокращение LIFO означает last in – first out (последний вошел — первый вышел)]. Это сокращение представляет удобный способ запомнить механизм работы стека.

Как в повседневной жизни, так и в программировании очередь — это способ организации ожидания. Очередь, которая строится в памяти ЭВМ, имеет аналоги в реальной жизни, например очередь, которая образуется перед кассой передвижного театра, те кто пришел раньше, стоят в очереди и постепенно обслуживаются билетным кассиром, а те, кто подошел только что, становят-

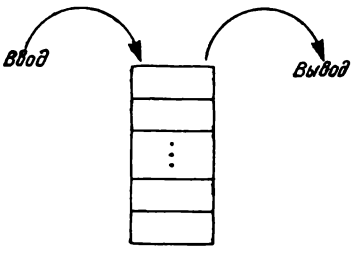


Рис. 17.1. Стек.

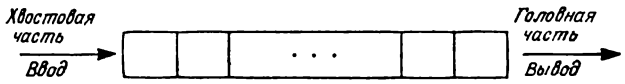



Рис. 17.2. Очередь.

ся в конец очереди. Подобным же образом можно различать два режима работы программы: первый состоит в получении данных и добавлении их в конец очереди и второй — в обработке данных, находящихся в начале очереди.

Таким образом, к данным, находящимся в очереди, можно получить доступ через две точки. Данные могут быть добавлены в конец очереди и удалены из ее начала. Элемент, который первым был добавлен в очередь, всегда первым достигает ее начала. Поэтому очередь можно описывать как структуру FIFO, что означает first in – first out (первый вошел — первый вышел).

Стеки и очереди могут быть реализованы в виде массивов. Стек с максимальным размером STACKSIZE может быть организован следующим образом:

```
const
    STACKSIZE = ...
type
    STACK = record
              OBJECT : array [1..STACKSIZE] of REAL;
              TOP : 0..STACKSIZE
            end;
var
    ONESTACK : STACK;
```

Каждая компонента массива ONESTACK.OBJECT является компонентой собственного стека. Так как стек может удлиняться и укорачиваться, необходим некоторый указатель для определения текущего размера стека. Поле ONESTACK.TOP выполняет эту функцию. Если ONESTACK пуст, то ONESTACK.TOP равен нулю. Если ONESTACK содержит от 1 до STACKSIZE объектов, то ONESTACK.TOP показывает на верхний элемент стека в пределах массива.

В примере 17.1 приведена процедура, которая выполняет операцию «занести» (push); другими словами, она добавляет элемент в вершину стека. Параметр NEW

**Пример 17.1. Добавление объекта в стек**

```
procedure PUSH (var NEWSTACK : STACK; NEWOBJECT :
        REAL);
(* Эта процедура добавляет значение NEWOBJECT в вер-
   шину NEWSTACK*)
begin
    if NEWSTACK.TOP = STACKSIZE then
        WRITELN ('Стек заполнен.')
    else
        begin
            NEWSTACK.TOP := NEWSTACK.TOP + 1;
            NEWSTACK.OBJECT[NEWSTACK.TOP] :=
                NEWOBJECT
        end
end; (* процедуры PUSH*)
```

OBJECT имеет тип REAL, так как объекты стека — типа REAL (если бы тип стека был определен по-другому, мы должны были бы работать с объектами соответствующего типа). Так как невозможно занести объект в стек, который уже заполнен, процедура проверяет это условие и при его выполнении сигнализирует об ошибке.

В примере 17.2 показана процедура, которая выполняет

**Пример 17.2. Удаление объекта из стека**

```
procedure POP  (var OLDSTACK : STACK; var OLDO-
        BJECT : REAL);
(* Эта процедура удаляет один объект из вершины
   OLDSTACK и помещает его значение в OLDO-
   BJECT.*)
begin
    if OLDSTACK.TOP = 0 then
        WRITELN ('Стек пуст.')
    else
        begin
            OLDOBJECT := OLDSTACK.OBJECT
            [OLDSTACK.TOP];
            OLDSTACK.TOP := OLDSTACK.TOP-1
      end
end; (* процедуры POP*)
```

обратную по отношению к «занести» операцию, а именно операцию «удалить» (pop). Удалить объект из стека — это значит переписать его из вершины стека. Каждый раз, когда вызывается процедура pop, указатель вершины стека уменьшается на единицу, чтобы показывать на новую вершину. Таким образом, когда из стека удалены все объекты, указатель приобретает значение 0, какое он и имел первоначально.

Когда с помощью массива реализуется очередь, приемы работы с этим объектом изменяются. Во-первых, необходимы указатели на первый и последний элементы. Очередь с максимальным размером QUEUESIZE может быть организована следующим образом:

```
const
      QUEUESIZE = ...
type
      QUEUE = record
                  OBJECT : array[0..QUEUESIZE]    of
                  REAL;
                  FRONT, REAR : 0..QUEUESIZE
                end;
var
      ONEQUEUE : QUEUE;
```

Характерным отличием в описании для STACK и QUEUE является то, что массив для QUEUE начинается с индекса 0, т. е. массив для QUEUE содержит QUEUESIZE + 1 компонент. Почему? Причина кроется в алгоритме, используемом для работы с очередью.

В примере 17.3 показана процедура, которая добавляет элемент в конец очереди. Поскольку начало и конец очереди непрерывно перемещаются в пределах массива, удобно представлять себе массив в виде кольца, как это показано на рис. 17.3. Для того чтобы обходить массив «по кольцу», используется операция mod. Если REAR меньше, чем QUEUESIZE-1, он увеличивается обычным способом. После же того, как REAR достигает значения QUEUESIZE-1, он сбрасывается в нуль. Это легко проверить: (QUEUESITE-1) + 1 равно QUEUESIZE, а QUEUESIZE mod QUEUESIZE равно нулю (как показано на рис. 17.3, в данной схеме OBJECT[QUEUESIZE] не используется).

Процедура, приведенная в примере 17.4, удаляет

**Пример 17.3. Добавление элемента в очередь**

```
procedure ADDQ (var NEWQUEUE : QUEUE;
                    NEWOBJECT : REAL);
(* Эта процедура добавляет значение NEWOBJECT
   в хвост NEWQUEUE.*)
begin
    with NEWQUEUE do
        begin
            REAR := (REAR +1) mod QUEUESIZE;
            if REAR = FRONT then
              WRITELN('Очередь полна.')
            else
              OBJECT[REAR] := NEWOBJECT
        end
end; (* процедуры ADDQ*)
```

элемент из начала очереди. Здесь мы сталкиваемся с кажущимся противоречием: каким образом состояние «пустая очередь» в DELETEQ может определяться идентично состоянию «очередь заполнена» в ADDQ? Но несмотря на то, что в обеих процедурах используются одинаковые булевы выражения, в каждой из них проверяются различные условия. ADDQ перед проверкой

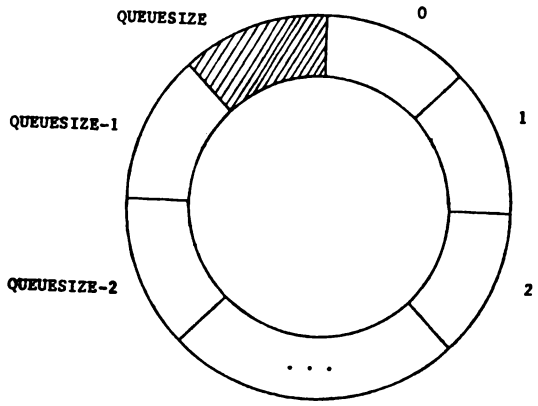



Рис. 17.3

увеличивает REAR, тем самым она проверяет, существует ли более одной свободной позиции между FRONT и REAR. Если существует только одна свободная позиция, очередь считается заполненной.

---

**Пример 17.4. Удаление элемента из очереди**

```
procedure DELETEQ (var OLDQUEUE : QUEUE; var
          OLDOBJECT : REAL);
(* Эта процедура выбирает элемент из начала очереди
   OLDQUEUE и присваивает его значение OLDOBJECT.*)
begin
        with OLDQUEUE do
             if REAR = FRONT then
                 WRITELN('Очередь пустая.')
             else
                 begin
                       FRONT := (FRONT + 1) mod QUE
                       UESIZE;
                       OLDOBJECT := OLDOBJECT
                       [FRONT]
                 end
end; (* процедуры DELETEQ*)
```

---

## 17.2. Переменные-указатели

*Переменная-указатель* — это такая переменная, которая «указывает» на другую переменную. Когда говорится, что переменная *X* указывает на переменную *Y*, это означает, что переменная *X* содержит адрес переменной *Y*. Приведем описание переменной-указателя:

```
var
   REALNUM :↑ REAL;
```

В пределах программы REALNUM может указывать на переменную типа REAL. (Вертикальная стрелка в описании показывает, что REALNUM — указатель.) Для того чтобы напечатать значение переменной, на которую показывает REALNUM, можно записать

```
WRITELN(REALNUM ↑)
```

REALNUM ↑ — это вещественное число, на которое указывает REALNUM. Можно использовать REALNUM ↑

точно так же, как и другие переменные типа REAL. Например, можно написать

```
REALNUM↑ := SQRT(2)
```

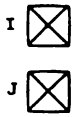
Откуда появилось REALNUM↑? Эта переменная не описывает вместе с REALNUM и не образуется автоматически. Для того чтобы создать переменную, на которую показывает REALNUM, нужно использовать процедуру NEW. Вот пример вызова процедуры NEW :

```
NEW(REALNUM)
```

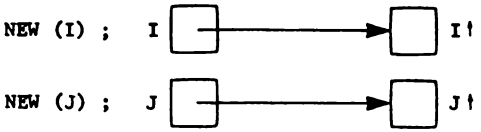
Когда этот оператор выполнится, будет создана новая переменная, и ее адрес будет помещен в REALNUM. Первоначально новая переменная не будет иметь значения. Приведем описание двух переменных-указателей:

```
var
  I, J :↑ INTEGER;
```

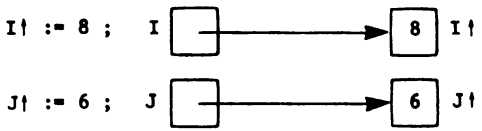
Вначале I↑ и J↑ не показывают ни на что. Их можно изобразить следующим образом:



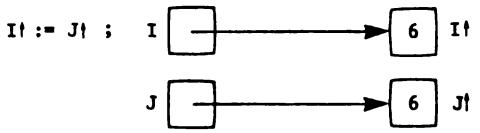

Следующая последовательность операторов записывает в I и J адреса, на которые они будут указывать:



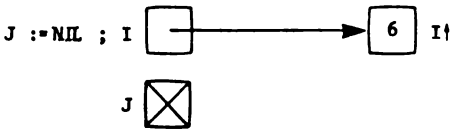

Теперь I↑ и J↑ можно присвоить значения:



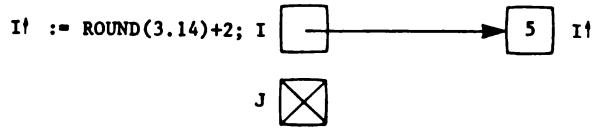

Если мы затем присвоим I↑ значение J↑, результат будет выглядеть следующим образом:



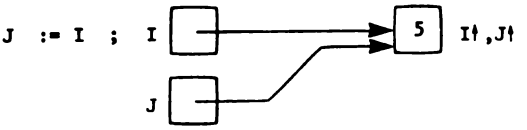

Этот процесс можно повернуть вспять. Присваивая переменной-указателю значение NIL, мы можем вернуть ее в исходное состояние. Когда указатель содержит значение NIL, он не показывает ни на что. (NIL — зарезервированный идентификатор языка Паскаль.) Вот что происходит, если J присвоено значение NIL:




Заметим, что теперь J↑ не определено. Присвоим I↑ другое значение:




Можно сделать J равным I, что приведет к следующему результату:




Теперь I и J содержат один и тот же адрес. Ячейка, на которую раньше указывало J, продолжает существовать; однако, поскольку на нее ничто не указывает, она более недоступна. Из приведенных выше примеров видно, что присваивания I := J и I↑ := J↑ дают существенно различный результат. Первое изменяет указатель, второе — значение целочисленной переменной.

Присваивания, выполняемые над указателями, должны подчиняться обычным правилам соответствия типов.

Важно заметить, что переменной-указателю не может быть присвоено значение другой переменной-указателя, если они обе не описаны идентично. Например, поскольку REALNUM имеет тип ↑REAL, а I — тип ↑INTEGER, нижеследующие операторы присваивания неверны:

```
REALNUM := I
I := REALNUM
```

Указатель, который указывает на переменную определенного типа данных, считается *ограниченным* этим типом. REALNUM ограничен типом REAL; I и J ограничены типом INTEGER. Указатель может быть ограничен любым типом (структурированным или неструктурированным), если только последний специфицирован как идентификатор типа в описаниях. Однако, когда указатель ограничен типом «запись с вариантами», оператор NEW может быть использован в несколько ином виде.

Допустим, имеются следующие описания:

```
type
    DOCUMENT = (STATUTE, HEARINGS, BRIEF);
    LEGAL_REFERENCE = record
                        case SOURCE : DOCUMENT of
                          STATUTE : (...);
                          HEARINGS : (...);
                          BRIEF : (...)
                        end;
var
    GERMAINE_THING : ↑LEGAL_REFERENCE;
```

Для того чтобы создать область, на которую будет указывать GERMAIN_THING, можно просто написать

```
NEW(GERMAIN_THING)
```

Этот оператор допускает использование любого определения записи. С другой стороны, если мы заранее знаем, какая из альтернатив будет использована, мы можем указать это во время вызова NEW. Зная, что будет использоваться только вариант STATUTE записи LEGAL_REFERENCE, можно написать

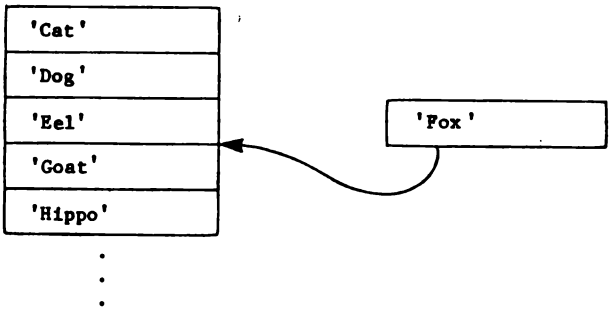
```
NEW(GERMAINE_THING, STATUTE)
```

Методы использования указателей и записей с вариантами достаточно распространены в литературе по программированию, поскольку оба этих средства позволяют программисту использовать некоторые внутренние механизмы ЭВМ. Однако за это, как и за большинство виртуозных приемов программирования, приходится расплачиваться потерей наглядности и увеличением сложности. Цель использования указателей и записей с вариантами — помочь программисту создавать типы данных, в которых «реализуется функция следования». В двух следующих параграфах будет показано, как, используя переменные-указатели, упростить разработку некоторых распространенных структур.

## 17.3. Связный список

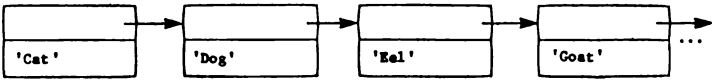
Допустим, необходимо хранить список названий, упорядоченных по алфавиту. Названия будут непрерывно добавляться и удаляться из списка. Какая структура данных подойдет для этой цели?

Для начала можно попробовать использовать массив строк. Когда появляется новая строка, она будет вставляться в массив на соответствующее место. Однако, чтобы освободить место для нового названия, программа должна передвинуть в массиве множество других названий на одну позицию массива:




Даже если программа оставляет свободные места для облегчения вставок, может оказаться так, что массив заполнится в одних местах и останется пустым — в других. Альтернативным является представление списка

названий с помощью структуры данных, называемой *связным списком.* Связный список можно изобразить следующим образом:




Каждый элемент связного списка представляет собой запись, составленную из двух полей. Первое поле содержит данные, второе является указателем, который указывает на следующий элемент. Вот описание связного списка:

```
type
     NAMELINK = ↑NAMELIST;
     NAMELIST = record
                      DATA : STRING;
                      NEXT : NAMELINK
                end;
var
     NAMES : NAMELINK;
```

NAMES указывает на начало списка. Двигаясь с помощью указателей, можно пройти последовательно по всем названиям (заметим, что описание NAMELINK включает в себя ссылку на NAMELIST, несмотря на то что NAMELIST еще не описан. Язык Паскаль допускает это исключение из обычного правила[1], по которому ссылки на идентификатор допускаются только после того, как он описан).

В примере 17.5 приведена процедура, которая распе-

---

**Пример 17.5. Проход связного списка**

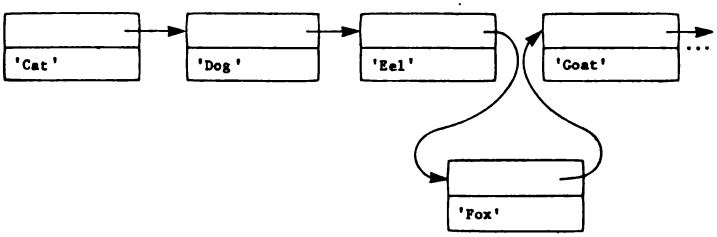
```
procedure PRINTLIST(BASE : NAMELINK);
begin
      while BASE <> NIL do
            begin
              WRITELN(BASE ↑ .DATA);
              BASE := BASE ↑ .NEXT
            end
end; (* процедуры PRINTLIST *)
```

---

[1] Это единственное исключение такого типа в языке. — *Прим. ред.*

чатывает элементы (*узлы*) связного списка. У процедуры один аргумент — указатель, который показывает на начало списка. Затем он заменяется следующим указателем, пока не достигает указателя со значением NIL. (Связный список обычно заканчивается указателем со значением NIL.)

Чтобы вставить узел в связный список, единственное, что нужно сделать, — это изменить несколько указателей. Сами узлы не должны перемещаться, т. е.:




Если переменная-указатель с именем EEL показывает на узел, содержащий 'Eel', и FOX указывает на узел, содержащий 'Fox', тогда вставка может быть выполнена с помощью следующей последовательности операторов:

```
FOX↑.NEXT := EEL↑.NEXT;
EEL↑.NEXT := FOX;
```

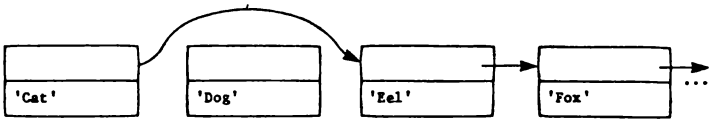
Теперь EEL↑ будет содержать 'Eel', EEL↑.NEXT↑ будет содержать 'Fox' и EEL↑.NEXT↑.NEXT↑ будет содержать 'Goat'.

Этот алгоритм вновь появляется в примере 17.6. Процедура ADDLIST принимает в качестве аргумента строку и вставляет ее в связный список. Если список был до выполнения этого действия пуст, ADDLIST возвращает список с одним узлом. Иначе список просматривается, пока не будет найдена подходящая позиция для строки. Если процедура достигнет конца списка и все еще не найдет строку, которая лексикографически «больше», чем вставляемая, тогда вставляемая строка будет добавлена к концу списка.

**Пример 17.6. Вставка узла в связный список**

```
procedure ADDLIST(NEWNAME : STRING; var BASE
                                        : NAMELIST);
var
    NEWNODE, LISTPOS : NAMELIST;
begin
    LISTPOS := BASE;
    NEW(NEWNODE);
    NEWNODE ↑ . DATA := NEWNAME;
    if BASE = NIL then (* список пуст? *)
        begin
            NEWNODE ↑ . NEXT := NIL;
            BASE := NEWNODE
        end
    else
        begin
            while (LISTPOS ↑ . DATA < NEWNAME) and
            (LISTPOS ↑ . NEXT <> NIL) do
                LISTPOS := LISTPOS ↑ . NEXT;
            NEWNODE ↑ . NEXT := LISTPOS ↑ . NEXT;
            LISTPOS ↑ . NEXT := NEWNODE
        end
end; (* процедуры ADDLIST *)
```

Удаление узла из связного списка осуществляется подобными манипуляциями. Допустим, NAMES показывает на узел, содержащий 'Cat'. Чтобы удалить узел, содержащий 'Dog', нужно изменить указатели следующим образом:




NAMES↑.NEXT↑:= NAMES↑.NEXT↑.NEXT

Для того чтобы выполнить удаление, нужен только один оператор. Узел, содержащий 'Dog', более не является частью списка, так как на него никто не указывает. Узел, содержащий 'Cat', теперь предшествует узлу, содержащему 'Eel'; таким образом, при удалении сохраняется алфавитный порядок.

## 17.4. Бинарные деревья

В связном списке каждый узел содержит указатель на другой узел. Бинарное дерево похоже на связный список, за тем исключением, что каждый узел содержит *два* указателя. Начальная точка бинарного дерева называется *корневым узлом.* Обычно бинарное дерево изображается с корневым узлом вверху.

Каждый из двух указателей узла дерева показывает на другое дерево. Один из них показывает на левое поддерево, а другой — на правое поддерево. На рис. 17.4

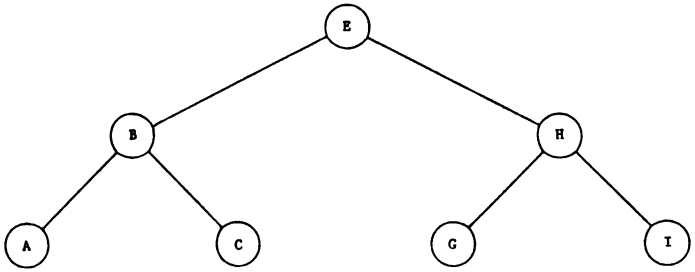



Рис. 17.4.

корневой узел *E* показывает на *B* и *H*. Узел *B* является корневым узлом для левого поддерева *E*; узел *H* является корневым узлом для правого поддерева *E*. Узлы *B* и *H* в свою очередь указывают на свои поддеревья. За исключением вершин самого нижнего яруса, каждый узел бинарного дерева имеет одно или два поддерева.

Бинарные деревья имеют множество применений. В последующих примерах мы будем использовать бинарные деревья для хранения множества строк в алфавитном порядке. Каждый узел бинарного дерева содержит единственную строку. Если левое поддерево определено, его корень будет содержать лексикографически меньшую строку. Если определено правое поддерево, оно будет содержать лексикографически бóльшую строку. Дерево такого специального вида называется *деревом бинарного поиска.* На рис. 17.5 приведено несколько примеров деревьев бинарного поиска.

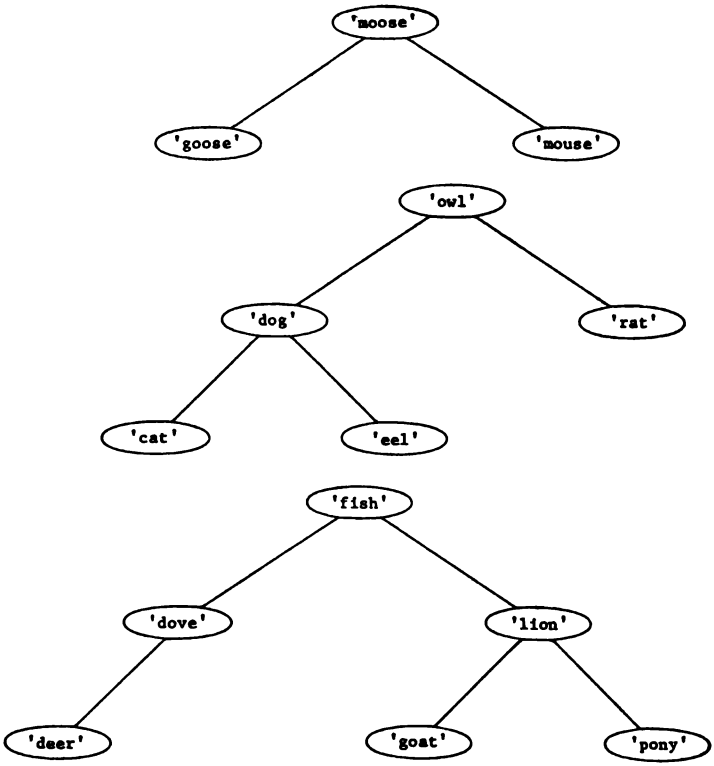



Рис. 17.5.

Приведем описание узла бинарного дерева:

```
type
    TREELINK = ↑TREE;
    TREE = record
             DATA : STRING;
             LEFT, RIGHT : TREELINK
           end;
```

Чтобы определить, содержится ли строка в каком-либо узле бинарного дерева, для упрощения поиска можно использовать следующее свойство. Вместо того чтобы исследовать каждую вершину дерева, можно начать с корня и затем следовать либо по левому указателю, либо по правому в зависимости от результата

последовательности лексикографических сравнений. Если узел, который мы исследуем, содержит строку, которая больше, чем наша строка, нужно идти налево, в противном случае нужно идти направо. В примере 17.7 показана функция, которая использует этот алгоритм.

---

**Пример 17.7 Поиск в двоичном дереве**

```
function SEARCHTREE(ROOT : TREELINK; OBJECT :
                                        STRING) : BOOLEAN;
(* Если дерево содержит OBJECT, функция возвращает
   значение TRUE, иначе FALSE.*)
begin
    SEARCHTREE := FALSE;
    while ROOT <> NIL do
          if ROOT ↑ .DATA = OBJECT then
              SEARCHTREE := TRUE
          else
              if ROOT ↑ .DATA > OBJECT then
                ROOT := ROOT ↑ .LEFT
              else
                ROOT := ROOT ↑ .RIGHT
end; (* функции SEARCHTREE *)
```

---

Подобный же алгоритм может быть использован для добавления левого узла к дереву. Новые узлы всегда

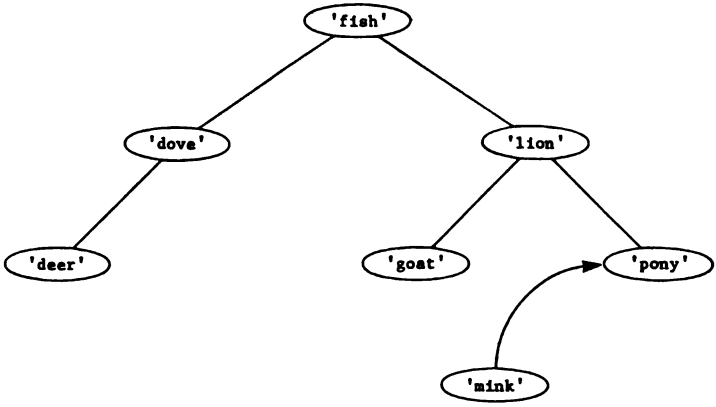



Рис. 17.6.

присоединяются к нижним вершинам дерева, так что дерево всегда растет вниз. На рис. 17.6 показан пример операции добавления.

Двигаясь по дереву, начиная из корня, мы видим, что 'mink' больше, чем 'fish', больше чем 'lion', и меньше, чем 'pony'. Следовательно, узел, содержащий 'mink', становится левым поддеревом узла, содержащего 'pony'. Функция, приведенная в примере 17.8, использует рекурсивный алгоритм для выполнения этой операции.

---

**Пример 17.8. Добавление узла к двоичному дереву**

```
function ADDTREE (BASE : TREELINK; NEWNODE :
                  STRING) : TREELINK;
begin
    if BASE = NIL then
        begin
            NEW(BASE);
            BASE ↑ . DATA := NEWNODE;
            BASE ↑ . LEFT := NIL;
            BASE ↑ . RIGHT := NIL
        end
    else
        if   BASE ↑ . DATA > NEWNODE then
            BASE ↑ . LEFT := ADDTREE(BASE ↑ . LEFT,
                                     NEWNODE)
        else
            BASE ↑ . RIGHT := ADDTREE(BASE ↑ . RIGHT,
                                      NEWNODE);
    ADDTREE := BASE
end; (* функции ADDTREE* )
```

---

Чтобы распечатать дерево бинарного поиска в относительном порядке, можно использовать алгоритм обхода, называемый *обратным обходом.* Простейший способ описать этот алгоритм — определить его рекурсивно. Для того чтобы совершить обратный обход дерева при заданном корне, программа должна совершить три шага: во-первых, выполнить обратный обход левого поддерева; во-вторых, напечатать содержимое корня; в-третьих, выполнить обратный обход правого поддерева. Когда программа обнаруживает, что указатель имеет значение NIL, то продвижение в направлении обхода в данном под-

дереве прекращается. В примере 17.9 показана рекурсивная процедура, которая печатает дерево бинарного поиска, используя обратный обход.

**Пример 17.9. Обход двоичного дерева**

```
procedure INORDER(BASE : TREELINK);
begin
    if BASE <> NIL then
        begin
          INORDER(BASE ↑ . LEFT);
          WRITELN(BASE ↑ . DATA);
          INORDER(BASE ↑ . RIGHT)
        end
end; (* процедуры  INORDER *)
```

Для обхода бинарного дерева могут быть использованы два других алгоритма. При прямом обходе содержимое корня печатается до обхода двух поддеревьев. При концевом обходе содержимое корневого узла печатается после того, как будет совершен обход двух поддеревьев. В случае поиска эти два обхода, в отличие от обратного не очень полезны. Однако в других применениях двоичных деревьев прямой или концевой обходы могут оказаться более предпочтительными.

ВОПРОСЫ

1. Напишите процедуру, которая удаляет узел из связного списка.

2. Напишите процедуру, которая строит обратный порядок узлов в связном списке.

3. Спроектируйте представление указателя для структуры «очередь».

а) Напишите процедуру, которая добавляет элемент в конец такой очереди.

б) Напишите процедуру, которая удаляет элемент из начала такой очереди.

*4. Напишите рекурсивную функцию, которая определяет, помещена ли строка в бинарное дерево поиска.

5. Напишите нерекурсивную процедуру, которая добавляет узел к бинарному дереву поиска.

*6. Напишите булеву функцию, которая определяет, равны ли два бинарных дерева.

7. Спроектируйте запись произвольной структуры с полями по вашему выбору. Определите конкретно, какое поле будет использоваться для определения отношения порядка между различными записями.

а) Напишите описание бинарного дерева, узлы которого содержат записи такого типа.

б) Напишите описания двух процедур: одной для добавления новых узлов и другой для печати содержимого выбранного узла.

*8. Напишите рекурсивную процедуру, которая выполняет концевой обход дерева.

# Глава 18
# Файлы

## 18.1. Обработка последовательных файлов

Как известно, текстовые файлы могут использоваться для хранения и последующего воспроизведения значений типа INTEGER, REAL, CHAR и массивов литер. Для выполнения файловых операций над значениями других типов в язык Паскаль введено понятие *файла*. Переменная типа файл, подобно переменной типа массив, обладает такой характеристикой, как тип компонент, которая описывает, что́ данный файл содержит. Файл может иметь любой тип компонент, за исключением файлового.

Описание файла выглядит следующим образом:

```
STORE _ SALES : file of REAL
MAILING _ LIST : file of record
                          NAME, STREET, CITY :
                                              STRING;
                          STATE : STRING[2];
                          ZIP _ CODE : INTEGER
                         end;
```

Файлы открываются и закрываются так же, как текстовые файлы. Эффект выполнения RESET, REWRITE и CLOSE не зависит от того, над какой файловой переменной эти действия выполняются: текстовым файлом или файлом. (Обычно файлы логически связаны с устройствами с блочной структурой). Различие в работе возникает на этапе доступа к хранимой в этих файлах информации[1].

При описании файлов автоматически создается вспомогательная переменная, которая называется *буферной*

[1] Вообще говоря, никакого отличия нет. Текстовые файлы являются частным случаев файлов. Проводимое здесь отличие носит методический характер и в этом смысле полностью оправданно. — *Прим. ред.*

*переменной* (или просто *буфером*) данного файла. Именем буферной переменной является просто имя файла, дополненное литерой "↑". Рассмотрим следующее описание:

```
type
AUTO = record
         MAKE, MODEL : STRING;
         YEAR : 1 .. 9999;
         LICENSE : STRING[6]
       end;
  FILETYPE = file of AUTO;
var
  VEHICLES : FILETYPE;
```

Конечно, идентификатором VEHICLES представлен новый файл. Буферная переменная для VEHICLES имеет имя VEHICLES↑. Таким образом, VEHICLES↑ является переменной типа AUTO. Несмотря на то что файловая переменная не может применяться в операторе присваивания, использование буферной переменной подчиняется общим правилам. Например, если файл содержит компоненты типа REAL, то буферная переменная данного файла может быть распечатана, использована в расчетах или изменена путем присвоения нового значения. Однако основное назначение буферной переменной заключается во временном хранении значений при записи или считывании их из файла.

Вызов процедуры RESET приводит к тому, что буферная переменная принимает значение первой компоненты открываемого файла. Для просмотра последующих компонент файла может использоваться стандартная процедура GET. Каждое применение GET помещает значение следующей компоненты файла в буфер и передвигает на одну позицию маркер файла. Для чтения следующей компоненты VEHICLES необходимо выполнить

```
GET(VEHICLES)
```

Окончание файла можно определить при помощи функции EOF. Если вызов GET приводит к тому, что значение функции EOF становится равным TRUE, то маркер файла достиг конца файла. В этом случае вызов GET делает значение буферной переменной не-

определенным. В примере 18.1 приводится программа, в которой читаются все компоненты (типа INTEGER) файла.

**Пример 18.1. Чтение файла**

```
program RETRIEVE;
var
    FILE _ NAME : STRING;
    DUMMY _ FILE : file of INTEGER;
begin
    WRITELN('Укажите имя внешнего файла');
    READLN(FILE _ NAME);
    RESET(DUMMY _ FILE, FILE _ NAME);
    WRITELN('В файле содержатся следующие значения:');
    while not EOF(DUMMY – FILE) do
        begin
            WRITELN(DUMMY _ FILE ↑); (* печать числа *)
            GET(DUMMY _ FILE); (* выборка следующего *)
        end
end.
```

Для записи значений в файл используется процедура PUT. Эта процедура помещает значение буферной переменной в компоненту файла, указанную маркером. После записи маркер передвигается на следующую позицию. Приведенные операторы обеспечивают запись одной компоненты файла VEHICLES:

```
VEHICLES ↑ . MAKE := 'Oldsmobile';
VEHICLES ↑ . MODEL := 'Cutlass';
VEHICLES ↑ . YEAR := 1979;
VEHICLES ↑ . LICENSE := 'ABX833';
PUT(VEHICLES);
```

Если маркер располагается где-то в середине файла, то PUT приводит к замене компоненты, помеченной маркером. С другой стороны, если маркер находится в конце файла, то выполнение PUT приводит к добавлению новой компоненты[1]. В примере 18.2 представлена про-

[1] Обычно считается, что записывать компоненты можно только в конец файла. — *Прим. ред.*

грамма, которая получает последовательность целых чисел и помещает их в указанный внешний файл. Если, скажем, пользователь вводит десять чисел, то после завершения программы внешний файл будет содержать десять компонент.

---

**Пример 18.2. Запись в файл**

```
program STORE;
var
    FILE_SIZE, COUNT : INTEGER;
    FILE_NAME : STRING;
    DUMMY_FILE : file of INTEGER;
begin
    WRITELN('Укажите имя внешнего файла.');
    READLN(FILE_NAME);
    WRITELN('Сколько чисел вы хотите записать в');
    WRITELN(FILE_NAME, '?');
    READLN(FILE_SIZE);
    REWRITE(DUMMY_FILE, FILE_SIZE);
    for COUNT := 1 to FILE_SIZE do
        begin
            READLN(DUMMY_FILE↑);
            PUT(DUMMY_FILE) (* записать число *)
        end;
    CLOSE(DUMMY_FILE, LOCK)
end.
```

---

Процедура, описанная в примере 18.3, получает в качестве аргументов два файла и выполняет копирование первого файла во второй. Процедура COPYFILE, используя RESET, прежде всего устанавливает маркер в начало файла SENDING, а затем при помощи REWRITE зачищает файл RECEIVING. (Предполагается, что перед вызовом процедуры файлы уже были открыты.) После открытия файлов в цикле *while* производится пересылка по одной компоненте содержимого файла SENDING в файл RECEIVING. Так как RESET автоматически выполняет операцию GET для первой компоненты, в цикле первоначально производится копирование, а потом считывание.

Для работы с текстовыми файлами могут использоваться процедуры GET и PUT. Просто текстовые файлы можно рассматривать как файлы с компонентами

типа CHAR. Эта точка зрения представляет в основном чисто теоретический интерес, так как процедуры текстовых файлов (т. е. READ, READLN, WRITE, WRITELN и PAGE) более полезны. Поскольку буферная переменная текстового файла имеет тип CHAR, то, используя

---

**Пример 18.3. Копирование файлов**

```
procedure COPYFILE(var SENDING, RECEIVING :
                        FILETYPE;
                        var COMPONENTS : INTEGER);
(* Эта процедура копирует содержимое SENDING в файл
   RECEIVING. Число скопированных компонент возвра-
   щается в COMPONENTS.*)
begin
    COMPONENTS := 0;
    RESET(SENDING);
    REWRITE(RECEIVING);
    while not EOF(SENDING) do
        begin
            RECEIVING ↑ := SENDING ↑ ;
            PUT(RECEIVING);
            GET(SENDING);
            COMPONENTS := COMPONENTS + 1
        end
end;            (* процедуры COPYFILE *)
```

---

GET или PUT, можно передать только одну литеру между буферной переменной и текстовым файлом. Даже для моделирования операции READ, считывающей одну литеру, при работе с текстовым файлом требуется по крайней мере два оператора. Например, действию READ(ONE_CHAR) эквивалентны следующие два оператора:

```
GET(INPUT);
ONE_CHAR := INPUT↑;
```

Следует отметить одно ограничение в использовании процедур при работе с файлами (и текстовыми в том числе): нельзя получить доступ к какой-либо компоненте файла, не прочитав всех предшествующих компонент. Другими словами, до сих пор рассматривался лишь *последовательный режим обработки.* Если необхо-

димо обработать все компоненты в порядке их расположения в файле, то последовательный режим обработки для этих целей полностью подходит. Однако если необходимо обработать какую-то одну компоненту, то в этом случае в языке Паскаль предусмотрены специальные средства, о которых речь пойдет в следующем разделе.

## 18.2. Режим произвольного доступа

Режим *произвольного доступа* — это такой способ обработки файла, при котором можно выполнять непосредственные обращения к любой компоненте файла. Режим произвольного доступа позволяет программе при обращении к некоторой компоненте не выполнять явно просмотр всех предыдущих компонент.

Каждая компонента файла имеет некоторый целочисленный идентификатор. Так как наиболее часто компоненты файла являются записями, то этот числовой идентификатор называется *номером записи*. Первая компонента файла имеет номер записи 0, вторая — номер записи 1 и т. д. Для перемещения маркера файла к заданной компоненте необходимо определить имя файла и номер записи этой компоненты. Для этих целей в языке Паскаль предусмотрена процедура SEEK. Предположим, что файловая переменная описана следующим образом:

```
STUDENTS : file of GRADE_RECORD
```

Для установки маркера файла STUDENTS на первую компоненту следует записать

```
SEEK(STUDENTS, 0)
```

Если необходимо переместить маркер к десятой компоненте, следует выполнить процедуру SEEK с такими параметрами:

```
SEEK(STUDENTS, 9)
```

---

SEEK
Общая форма записи:

SEEK (*файловая переменная, номер записи*)

SEEK перемещает маркер к указанной компоненте. Файловая переменная не может быть текстовым файлом.

После каждого вызова SEEK должны выполняться операции GET или PUT.

**Примеры:**

SEEK(VEHICLES, CAR_NUMBER div 2)
SEEK(PAYROLL, 30)

---

Выполнив позиционирование маркера, можно использовать GET или PUT для того, чтобы прочитать или заменить значение компоненты. Если ENROLLEE является переменной типа GRADE_RECORD, то вполне возможно записать значение ENROLLEE в четвертую компоненту файла STUDENTS. Делается это следующим образом:

```
SEEK(STUDENTS, 3);
STUDENTS↑ := ENROLLEE;
PUT(STUDENTS);
```

Для считывания компоненты с номером записи REC_NUMBER и присвоения полученного значения переменной APPLICANT следует выполнить такую последовательность операторов:

```
SEEK(STUDENTS, REC_NUMBER);
GET(STUDENTS);
APPLICANT := STUDENTS↑;
```

Для того чтобы в режиме произвольного доступа считать и затем изменить значение некоторой компоненты, следует выполнить два вызова процедуры SEEK. Один вызов для позиционирования маркера перед операцией GET, а другой для позиционирования маркера перед операцией PUT. Так как GET переместит маркер к следующей компоненте, то для изменения значения считанной компоненты следует возвратить маркер в прежнее положение. Приведенная последовательность операторов модифицирует компоненту с номером записи 14 в файле STUDENTS:

```
SEEK(STUDENTS, 14);
GET(STUDENTS);
STUDENTS↑.PROJECT := STUDENTS↑.PROJECT + 5;
SEEK(STUDENTS, 14);
PUT(STUDENTS);
```

Программа, представленная в примере 18.4, сочетает

режим последовательной обработки с режимом произвольного доступа. В программе выполняется последовательный просмотр файла. Компоненты, которые необходимо изменить, корректируются в режиме произвольного доступа. Произвольный доступ необходим для выполнения корректировки сразу после операции чтения. Без использования SEEK невозможно возвратить маркер к предыдущей компоненте файла, кроме как переместив маркер в начало файла при помощи RESET.

---

**Пример 18.4. Корректировка файла**

```
program UPDATELIMIT;
(*Эта программа проверяет состояние кредита всех счетов, хранящихся в файле. Если ход кредитования по некоторому счету идет нормально, то верхняя граница кредита будет скорректирована. *).
type
    ACCOUNT = record
                        NAME, STREET, CITY :
                        STRING[20];
                        STATE : packed array [1 .. 2]
                        of CHAR;
                        ZIPCODE : INTEGER;
                        BALANCE : REAL;
                        LIMIT : REAL;
                        RATING : (GOOD, FAIR,
                        POOR)
                    end;
var
    ACCOUNT_FILE : file of ACCOUNT;
    NEW_LIMIT : REAL;
    REC_NUMBER : INTEGER;
begin
    RESET(ACCOUNT_FILE, 'ACCOUNTS.DATA');
    (*--получить новую верхнюю границу--*)
    WRITELN('Какова новая верхняя граница кредита')
    WRITELN('для счетов с хорошими показателями?')
    READLN(NEW_LIMIT);
    (*--просмотреть записи--*)
    REC_NUMBER := 0;
    while not EOF(ACCOUNT_FILE) do
        begin
            if ACCOUNT_FILE↑.RATING = GOOD
                                        then
```

```
                begin
                    SEEK(ACCOUNT_FILE,
                                    REC_NUMBER);
            (* возврат к считанной компоненте *)
                    ACCOUNT_FILE↑.LIMIT
                                    := NEW_LIMIT;
                    PUT(ACCOUNT_FILE)
                end;
            GET(ACCOUNT_FILE);
            REC_NUMBER := REC_NUMBER + 1
        end
end.
```

---

Произвольный доступ также облегчает удаление компонент. Чтобы убрать некоторую компоненту из файла, следует выполнить либо логическое, либо физическое удаление. Логическое удаление компоненты предполагает установку значений некоторых индикаторов, которые указывают пассивное состояние компоненты. В противоположность этому физическое удаление связано с возвращением для повторного использования пространства, занимаемого удаленной записью.

Логическое удаление требует, чтобы тип компонент включал вспомогательное поле индикатора. При чтении компоненты из файла следует проверять это поле индикатора, чтобы определить, является ли компонента удаленной или нет. Предположим, имеется такое описание:

```
ROSTER : file of
            record
              NAME : STRING;
              ACTIVE : BOOLEAN
            end;
```

Если ACTIVE имеет значение TRUE, то компонента продолжает использоваться. При значении FALSE программа должна рассматривать данную запись как отсутствующую в файле. Например, чтобы удалить компоненту с номером записи 10, следует выполнить такую последовательность операторов:

```
SEEK(ROSTER, 10);
ROSTER↑.ACTIVE := FALSE;
PUT(ROSTER);
```

Чтобы последовательно распечатать все имена файла ROSTER, следует записать

```
RESET(ROSTER);
while not EOF(ROSTER) do
      begin
            if ROSTER ↑ .ACTIVE then
               WRITELN(ROSTER ↑ .NAME);
            GET(ROSTER)
      end;
```

В примере 18.5 описана процедура, которая выполняет физическое удаление. Начиная с компоненты следующей за удаляемой, данная процедура перемещает все компоненты на место предыдущей компоненты. Таким образом компонента, которая должна быть удалена, замещается следующей компонентой и т. д. (После завершения цикла компонента с номером записи REC_NUMBER будет соответствовать компоненте с номером записи REC_NUMBER + 1 в первоначальном файле.) Данная процедура позицирует маркер таким образом, что закрытие файла в режиме CRUNCH отсекает последнюю компоненту первоначального файла. На рис. 18.1 показана последовательность состояний для файла из семи компонент при удалении третьей компоненты.

---

**Пример 18.5. Физическое удаление компоненты файла**

```
procedure REMOVE(REC_NUMBER : INTEGER;
                 var THE_FILE : ACCOUNT);
(* Эта процедура удаляет компоненту с номером записи
  REC_NUMBER из заданной файловой переменной.*)
var
    COUNT : INTEGER;
begin
    COUNT := REC_NUMBER;
    SEEK(THE_FILE, REC_NUMBER + 1);
    GET(THE_FILE);
    while not EOF(THE_FILE) do
        begin
            SEEK(THE_FILE, COUNT); (* возврат на
                                      одну компоненту *)
            PUT(THE_FILE);
            COUNT := COUNT + 1;
            SEEK(THE_FILE, COUNT + 1);
```

```
            GET(THE _ FILE);
        end;
    SEEK(THE _ FILE, COUNT - 1);
    GET(THE _ FILE);
    CLOSE(THE _ FILE, CRUNCH)
end; (* процедуры REMOVE*)
```

В примере 18.4 предполагалась необходимость коррекции только одного счета, а не всех. Как можно узнать, какая компонента содержит баланс счета разгневанного покупателя? Один из способов решения этой задачи связан с последовательным просмотром всего файла до тех пор, пока не будет найдена компонента с соответствующим полем имени. Однако если заранее известен

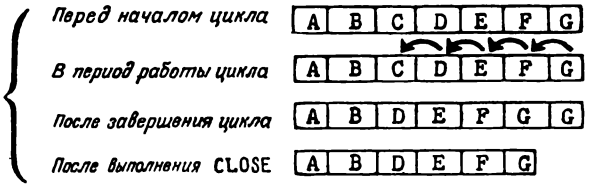



Рис. 18.1.

номер соответствующей записи, то можно более полно использовать возможности произвольного доступа. В конце концов для каждого покупателя можно задать код его счета, по которому легко найти номер записи.

Соглашения подобного рода подходят для многих приложений, но не для всех. Если счета часто добавляются и удаляются, жесткое соответствие между номером записи и кодом счета может оказаться нецелесообразным. В следующем разделе будет рассмотрен некоторый альтернативный подход к проблеме размещения компонент файла.

## 18.3. Связные списки в файлах

Использование произвольного доступа позволяет рассматривать файлы как массивы. В частности, механизм произвольного доступа позволяет обрабатывать «логиче-

ски» последовательные компоненты так, как будто они являются и физически последовательными. Есть ли разница между этими понятиями? Разве не предполагается, что если данные о поставщике № 1023 помещены в записи № 57, то данные о поставщике № 1024 помещаются в записи № 58? Ответом будет «не обязательно». В случае больших файлов с частыми вставками и удалениями требование хранить компоненты строго последовательно приводит к большим накладным расходам.

Можно упростить вставку и удаление, если использовать некоторые методы, описанные в гл. 17. Более того, можно отказаться от необходимости введения внешних кодовых чисел, соответствующих номерам записей. Это возможно, если организовать файлы не как массивы, а как динамические структуры. В данном случае нельзя использовать переменные-указатели, как в гл. 17, но можно воспользоваться аналогичным методом. Поскольку номера записей — целые числа, можно определить поля типа INTEGER, которые будут использоваться как указатели на другие компоненты. Вот пример описания файла, организованного в виде списка:

```
type
    NODE = record
             DATA : ...;
             NEXT : INTEGER
           end;
var
    LIST_FILE : file of NODE;
```

В каждой компоненте LIST_FILE содержится поле NEXT, содержащее номер следующей в списке записи. Поскольку наименьший номер записи в файле всегда 0, можно установить невозможное значение –1 в качестве признака конца списка. (Когда используются настоящие переменные-указатели, эту функцию выполняет идентификатор NIL.)

Описывая файл, нужно отметить некоторую компоненту, чтобы иметь возможность указывать на начало списка. Можно условно принять, что запись с номером 0 будет головной и фактически «пустой» записью. Поле DATA записи 0 игнорируется, а NEXT указывает на компоненту, которая в настоящий момент является первым узлом списка. Вот последовательность операторов, которая инициализирует LIST_FILE:

```
REWRITE(LIST_FILE);
LIST_FILE↑.NEXT := -1;
PUT(LIST_FILE);
```

Поле, которое используется для определения упорядоченности компонентов, называется *полем ключа.* Поскольку компоненты упорядочены в соответствии с содержимым их поля ключа, можно указывать конкретную запись, задавая значение ключа, который нужно отыскать. Допустим, имеются следующие описания:

```
type
    READER = record
                 NAME, STREET, CITY : STRING;
                 STATE : STRING[2];
                 ZIP : INTEGER;
                 EXPIRES : record
                               MONTH, YEAR :
                                         INTEGER
                           end;
                 NEXT : INTEGER
             end;
    SUBSCRIBERS = file of READER;
```

В примере 18.6 приведена процедура, имеющая параметрами имя файла и значение ключа. Процедура возвращает в качестве результата порядковый номер записи той компоненты, которая содержит заданное значение ключа. NAME является полем ключа. (В реальной практике, конечно, чтобы избежать одинаковых значений ключей, обычно выбирают поле ключа иначе). Предполагается, что, когда процедура вызывается, файл открыт.

Для завершения процедуры, когда список пуст или найден искомый ключ, используется EXIT. Алгоритм, представленный в примере, можно применить также и для связных списков, приводившихся в гл. 17. Укажем последовательность операторов, которая увеличивает поле года записи с ключом 'Donald Jones' на единицу.

```
FILE_SEARCH(MAG_FILE, 'Donald Jones', REC_
                                      NUMBER);
if REC_NUMBER = -1 then
    WRITELN('Запись не найдена.')
else
    begin
```

```
    SEEK(MAG_FILE, REC_NUMBER);
    GET(MAG_FILE);
    MAG_FILE↑.EXPIRES.YEAR := MAG_FILE↑.
                              EXPIRES.YEAR + 1;
    SEEK(MAG_FILE, REC_NUMBER);
    PUT(MAG_FILE)
end;
```

**Пример 18.6 Поиск в файле со структурой связного списка**

```
procedure FILE_SEARCH(var THE_FILE :
                          SUBSCRIBERS;
                      KEY : STRING;
                      var POSITION : INTEGER);
(* Эта процедура пытается установить положение компоненты, содержащей указанное имя. Если компонента найдена, в POSITION помещается номер соответствующей записи, иначе в POSITION помещается -1.*)
const
    NULL = -1;
begin
    RESET(THE_FILE);
    if THE_FILE↑.NEXT = NULL then (* файл пуст *)
        begin
            POSITION := NULL;
            EXIT(FILE - SEARCH)
        end;
    (*--чтение первой компоненты--*)
    POSITION := THE_FILE↑.NEXT;
    SEEK(THE_FILE, POSITION);
    GET(THE_FILE);
    (*--проход по списку--*)
    while THE_FILE↑.NEXT <> NULL do
        if THE_FILE↑.NAME = KEY then
            EXIT(SEARCH.FILE) (*-запись найдена-*)
        else
          begin
              POSITION := THE_FILE↑.NEXT;
              SEEK(THE_FILE, POSITION);
        end;  GET(THE_FILE)
    (*--если мы дошли так далеко, KEY в файле
                                    отсутствует--*)
    POSITION := NULL
end; (* процедуры SEARCH_FILE *)
```

Процедура, приведенная в примере 18.7, добавляет компоненту в конец файла; она модифицирует также соответствующее поле указателя таким образом, что добавленная компонента включается в связный список. После достижения конца файла процедура проверяет содержимое записи 0, чтобы установить, не пуст ли список. Если список не пуст, процедура перемещается по нему, пока не найдет нужную позицию для помещения новой компоненты.

---

**Пример. 18.7. Включение элемента в файл со структурой связного списка**

```
procedure FILE_INSERT(var THE_FILE : SUBSCRIBERS;
                      NEW_NODE : READER);
const
    NULL = -1;
var
    FINAL, POSITION : INTEGER;
begin
    (*--Поиск конца файла--*)
    FINAL := 0;
    RESET(THE_FILE);
    while not EOF(THE_FILE) do
        begin
            GET(THE_FILE);
            FINAL := FINAL + 1
        end.
    (*--Куда следует поместить в списке NEW_
                                    NODE?--*)
    SEEK(THE_FILE, 0);
    GET(THE_FILE);
    if THE_FILE↑.NEXT = NULL then
        POSITION := 0
    else
        POSITION := THE_FILE↑.NEXT;
    while THE_FILE↑ do
        while (NEXT <> NULL) and (NAME < NEW_
                                  NODE.NAME) do
            begin
                POSITION := NEXT;
                SEEK(THE_FILE, POSITION);
                GET(THE_FILE);
            end;
```

```
    (* -- Замена связей -- *)
    NEW_NODE.NEXT := THE_FILE↑.NEXT;
    THE_FILE↑.NEXT := FINAL;
    SEEK(THE_FILE, POSITION);
    PUT(THE_FILE);
    (* -- Поместить NEW_NODE в конец списка -- *)
    SEEK(THE_FILE, FINAL_1);
    GET(THE_FILE);
    THE_FILE↑ := NEW_NODE;
    PUT(THE_FILE)
end; (* процедуры INSERT_FILE *)
```

---

При вставке никакие компоненты файла не перемещаются, а только изменяются указатели. Следовательно, «логический адрес» компоненты определяется ее положением в списке, которое может совершенно отличаться от физического адреса. Положение компоненты в списке определяется значением ее поля ключа. На рис. 18.2 показаны изменения указателей, которые производятся процедурой в процессе вставки компоненты.

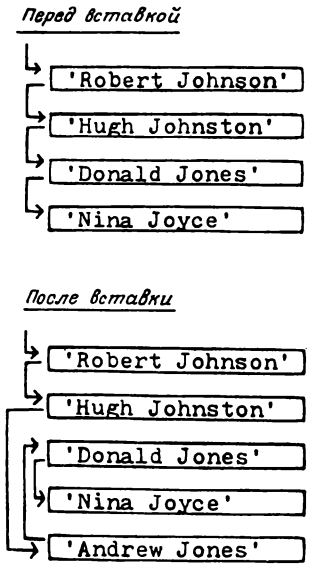



Рис. 18.2.

Можно ускорить вставку и поиск компонент, разбивая файл на несколько списков. Вместо того чтобы рассматривать файл как один связный список, можно интерпретировать его как набор связных списков, каждый из которых ассоциирован с группой возможных значений ключей. Например, для хранения компонент типа READER можно создать двадцать шесть связных списков, каждый из которых содержит компоненты, чьи поля имени начинаются с определенной буквы. Для того чтобы сделать это, нужно создать такую первую компоненту файла, которая содержала бы не один, а двадцать шесть ука-

зателей. Другими словами, нужно, чтобы первая компонента имела следующий тип:

```
type
    MAP = packed array ['A' .. 'Z'] of INTEGER;
```

В то же время хотелось бы, как и раньше, сохранить тот же тип компонент READER. Поскольку все компоненты файла должны быть одного типа, может показаться, что это представляет проблему. Эту проблему можно тем не менее преодолеть, используя в качестве компонент файла записи с вариантами. Новое описание SUBSCRIBERS можно записать следующим образом:

```
SUBSCRIBERS = file of
                  record
                    case BOOLEAN of
                      TRUE : (FIRST : MAP);
                      FALSE : (REST : READER)
                  end;
```

Здесь нет необходимости в использовании поля признака, так как вариант MAP используется только в записи с номером 0. Если, например, задано имя, начинающееся с 'W', то компонента файла с этим именем может быть найдена с помощью прохода по одному из двадцати шести списков — списка, включающего в себя имена, начинающиеся с 'W'. Чтобы найти начало этого списка, нужно выполнить следующие операторы:

```
SEEK(MY_FILE, 0);
GET(MY_FILE);
POSITION := MY_FILE↑.FIRST['W'];
```

Чтобы пройти этот список, используется тот же метод, что и прежде:

```
while POSITION <> -1 do
    begin
        SEEK(MY_FILE, POSITION);
        GET(MY_FILE);
          .
          .
          .
        POSITION := MY_FILE↑.REST.NEXT
end;
```

Эти структуры связных списков являются, конечно, только примерами структур файлов, которые здесь можно привести. Другие структуры могли бы основываться на очередях или бинарных деревьях или, возможно, на чем-либо другом. Используя режим произвольного доступа языка Паскаль, можно формировать такие структуры, которые удовлетворяют специальным требованиям в любых приложениях.

ВОПРОСЫ

*1. Каков результат следующего цикла?

```
while INPUT↑ <> '?' do
      GET(INPUT);
```

*2. Какие из приведенных определений правомерны:
   а) файлы записей;
   б) файлы вариантных записей;
   в) файлы массивов;
   г) массивы файлов;
   д) файлы множеств;
   е) множества файлов?

3. Даны два файла, типы компонент которых INTEGER. Компоненты файлов упорядочены по возрастанию. Напишите программу, которая осуществляет слияние этих двух файлов в третий, который тоже упорядочен по возрастанию. Например:

| Файл № 1 | Файл № 2 | Результирующий файл |
|---|---|---|
| 1 | –2 | –2 |
| 3 | 0 | 0 |
| 7 | 2 | 1 |
| 15 | 3 | 2 |
| | 4 | 3 |
| | 5 | 4 |
| | 10 | 5 |
| | | 7 |
| | | 10 |
| | | 15 |

4. Напишите программу обработки списка почтовых адресов, используя последовательный метод доступа. Программа должна обладать способностью добавления

и логического удаления записей. Пользователь также должен иметь возможность получать список всех адресов в определенном диапазоне почтовых индексов.

5. Напишите процедуру удаления компоненты с заданным значением ключа из файловой переменной типа SUBSCRIBERS.

6. Напишите программу, которая «улучшает» файл, организованный как связный список так, что его логически последовательные компоненты становятся также и физически последовательными.

7. Напишите новые описания для SUBSCRIBERS, FILE_SEARCH и FILE_INSERT, которые включали бы два поля ключа: NAME и ZIP. (Другими словами, соответствующие процедуры должны поддерживать два поля указателя в каждой компоненте. Один указатель указывает на компоненту с большим именем, другой указывает на компоненту с большим почтовым индексом.)

8. Напишите новые описания для SUBSCRIBERS, FILE_SEARCH и FILE_INSERT, которые работают с файлами, организованными как бинарные деревья.

# Приложения

## Приложение А. Зарезервированные слова

| | | |
|---|---|---|
| and | goto | record |
| array | if | repeat |
| begin | implementation | segment |
| case | in | separate |
| const | interface | set |
| div | label | then |
| do | mod | to |
| downto | not | type |
| else | of | unit |
| end | or | until |
| file | packed | uses |
| for | process | var |
| forward | procedure | while |
| function | program | with |

## Приложение Б. Стандартные идентификаторы

Приведенный ниже список содержит идентификаторы, используемые в версиях I—IV языка Паскаль. Стандартные функции и процедуры, заложенные в язык его разработчиками, называют *встроенными.* Поскольку существуют различия между версиями языка Паскаль, некоторые встроенные функции или процедуры из этого списка могут отсутствовать в конкретной версии языка. (Однако встроенные функции и процедуры, описанные в данной книге, существуют во всех версиях.)

- **Константы:** FALSE, MAXINT, NIL, TRUE
- **Типы:** BOOLEAN, CHAR, INTEGER, INTERACTIVE, PROCESSID, REAL, SEMAPHORE, STRING, TEXT
- **Файловые переменные:** INPUT, KEYBOARD, OUTPUT
- **Фукнции:** ABS, ARCTAN, ATAN, BLOCKREAD, BLOCKWRITE, CHR, CONCAT, COPY, COS, EOF, EOLN, EXP, IORESULT, LENGTH, LN, LOG, MEMAVAIL, ODD, ORD, POS, PRED, PWROFTEN, ROUND, SCAN, SIN, SIZEOF, SQR, SQRT, SUCC, TREESEARCH, TRUNC, UNITBUSY

• **Процедуры:** ATTACH, CLOSE, DELETE, DISPOSE, EXIT, FILLCHAR, GET, GOTOXY, HALT, IDSEARCH, INSERT, MARK, MEMLOCK, MEMSWAP, MOVELEFT, MOVERIGHT, NEW, OPENNEW, OPENOLD, PAGE, PMACHINE, PUT, READ, READLN, RELEASE, RESET, REWRITE, SEEK, SEMINIT, SIGNAL, START, STR, TIME, UNITCLEAR, UNITREAD, UNITWAIT, UNITWRITE, WAIT, WRITE, WRITELN

## Приложение B. Код ASCII

Множество литер ASCII включает в себя 128 элементов, из которых 95 могут быть представлены определенным типографским знаком. Остальные называются *управляющими литерами.* Управляющая литера «невидима» в том смысле, что она не ассоциируется с каким-либо типографским знаком. Большинство управляющих литер формируются на клавиатуре терминала с помощью одновременного нажатия клавиши *ctrl* и какой-либо другой клавиши.

| Порядковый номер | Литера | Порядковый номер | Литера |
|---|---|---|---|
| 0 | null | 25 | ctrl / Y |
| 1 | ctrl / A | 26 | ctrl / Z |
| 2 | ctrl / B | 27 | переход на другой регистр |
| 3 | ctrl / C | 28 | незадействовано |
| 4 | ctrl / D | 29 | незадействовано |
| 5 | ctrl / E | 30 | незадействовано |
| 6 | ctrl / F | 31 | незадействовано |
| 7 | ctrl / G [звонок] | 32 | пропуск |
| 8 | ctrl / H | | |
| 9 | ctrl / I | 33 | ! |
| 10 | ctrl / J [конец строки] | 34 | " |
| 11 | ctrl / K | 35 | # |
| 12 | ctrl / L [конец страницы] | 36 | $ |
| 13 | ctrl / M [возврат каретки] | 37 | % |
| 14 | ctrl / N | 38 | & |
| 15 | ctrl / O | 39 | ' |
| 16 | ctrl / P | 40 | ( |
| 17 | ctrl / Q | 41 | ) |
| 18 | ctrl / R | 42 | * |
| 19 | ctrl / S | 43 | + |
| 20 | ctrl / T | 44 | , |
| 21 | ctrl / U | 45 | - |
| 22 | ctrl / V | 46 | |
| 23 | ctrl / W | 47 | / |
| 24 | ctrl / X | 48 | 0 |

| Порядковый номер | Литера | Порядковый номер | Литера |
|---|---|---|---|
| 49 | 1 | . | . |
| 50 | 2 | 88 | X |
| 51 | 3 | 89 | Y |
| 52 | 4 | 90 | Z |
| 53 | 5 | 91 | [ |
| 54 | 6 | 92 | \ |
| 55 | 7 | 93 | ] |
| 56 | 8 | 94 | ↑ |
| 57 | 9 | 95 | — |
| 58 | : | 96 | ` |
| 59 | ; | 97 | a |
| 60 | < | 98 | b |
| 61 | = | 99 | c |
| 62 | > | . | . |
| 63 | ? | . | . |
| 64 | @ | . | . |
| 65 | A | 120 | x |
| 66 | B | 121 | y |
| 67 | C | 122 | z |
| 68 | D | 123 | { |
| 69 | E | 124 | ▪ |
| 70 | F | 125 | } |
| . | . | 126 | ~ |
| . | . | 127 | Вычеркивание / ожидание |

## Приложение Г. Целые с увеличенной точностью

В языке Паскаль имеется тип данных, называемый *длинное целое*, который позволяет хранить и обрабатывать большие целые значения. Каждая длинная целочисленная переменная получает атрибут длины в момент описания. Атрибут длины переменной типа длинное целое указывает максимальное число цифр, которое это число может содержать. Длинные целые могут иметь значение атрибута длины от одного до тридцати шести. Ниже приведены примеры описания некоторых переменных типа длинное целое:

```
LONG : INTEGER[12];
EXTRA_LONG : INTEGER[36];
```

Длинные целые числа могут появляться в теле программы и в описании констант. Целые, которые расположены вне диапазона –MAXINT..MAXINT (т. е. –32767..32767), представляются как длинные целые. Важно знать, что длинные целые не считаются значениями того же

типа, что и значения типа INTEGER. Между ними существует такое же соотношение, как и между значениями типов REAL и INTEGER. Переменным типа длинное целое могут быть присвоены обычные целые значения, но обратного действия выполнять нельзя. Для того чтобы преобразовать длинное целое в значение типа INTEGER, нужно использовать функцию преобразования типа TRUNC.

К длинным целым можно применять операции булевого сравнения. Над длинными целыми могут быть выполнены все операции целочисленной арифметики, за исключением *mod*. В арифметическом выражении можно применять как обычные, так и длинные целые; результатом будет длинное целое. Все операции над файлами вроде READLN или WRITELN могут использовать длинные целые. Для преобразования длинного целого в строку можно использовать процедуру STR.

Подобно значениям типа REAL, длинные целые не относятся к порядковым типам. Следовательно, длинное целое нельзя использовать в качестве управляющей переменной цикла *for* или селектора оператора *case*. Длинное целое можно передавать подпрограмме, но только в том случае, если ее атрибут длины не превышает атрибута длины соответствующего параметра. Функция, определяемая программистом, не может возвращать в качестве результата длинное целое.

## Приложение Д. Управление курсором с помощью GOTOXY

Если пользовательский терминал ввода-вывода (т. е. устройство 'CONSOLE:') является дисплеем, то для позиционирования курсора может быть использована встроенная процедура GOTOXY (Курсор — это символ, который показывает, где появится следующая выводимая литера.) Вызовы GOTOXY имеют следующий вид:

GOTOXY(COLUMN, ROW)

COLUMN и ROW — целые выражения, показывающие желаемую позицию курсора. Колонка 0 является самой левой на экране; строка 0 — самая верхняя строка экрана. На терминале, который отображает 24 строки и 80 позиций в строке, можно переместить курсор в правый нижний угол экрана с помощью вызова

GOTOXY(79,23);

Важно, что GOTOXY не стирает литер, которые уже выведены на экран. Большинство терминалов выполняет операцию «очистить экран» при получении определенной последовательности управляющих литер. Эта последовательность различна для разных моделей терминалов.

## Приложение Е. Ответы на вопросы, отмеченные '*'

### Глава 2

1. 
```
program TRIANGLE;
var
    BASE, HEIGHT : INTEGER;
begin
    BASE := 3;
    HEIGHT := 5;
    WRITELN('Основание треугольника равно ', BASE,
                                          ' см.');
    WRITELN('Высота треугольника равна ', HEIGHT,
                                          ' см.');
    WRITELN('Площадь треугольника равна ',.5
                                  *BASE*HEIGHT);
    WRITELN('Квадратных сантиметров.')
end.
```

3. 
```
A := L * W
M := R * SQR(R) div SQR(P)
S := (Y1 - Y2) div (X1 - X2)
X := ABS(A)
```

4. Программа содержит пять ошибок, каждой из которых достаточно, чтобы программа не выполнялась:
   а) Оператор *program* не заканчивается точкой с запятой.
   б) BEGIN является неправильным идентификатором, поскольку это зарезервированное слово.
   в) Вслед за зарезервированным словом *begin* не должна следовать точка с запятой.
   г) Для открывающей скобки оператора WRITELN нет соответствующей закрывающей скобки.
   д) Переменная TEMP не описана.

### Глава 3

1. Важно не путать WRITE и WRITELN. Их действия несколько различны.

3. program MORESUMS;
```
var
    ADDEND1, ADDEND2 : INTEGER;
begin
    WRITELN('Какие два числа вы хотите сложить?')
    READ(ADDEND1, ADDEND2);
    WRITELN('Сумма ', ADDEND1 : 1,' и ',
                                ADDEND2 : 1);
    WRITELN('равна ', (ADDEND1 + ADDEND2) : 1)
end .
```

**Глава 4**

1. Четвертый и пятый операторы присваивания неверны, так как SIN и COS всегда дают вещественный результат. Седьмой оператор присваивания неверен, так как использовано не целочисленное, а вещественное деление. Другие присваивания допустимы.
2. 17.693
   17.69
   17.7
   17.7
   1.769E+01
   1.769E+01
3. MARRIED and not MALE
   MALE and not MARRIED
   BLOND and not MARRIED
   not (MALE or MARRIED or EMPLOYED)
   not MARRIED or not EMPLOYED

**Глава 5**

1. В цикле бесконечно будет печататься апостроф.

2.
```
if X > 0 then
    if LN(X) > - 0 then
    WRITELN('X в степени Y равен ',
                            EXP(Y * LN(X)) : 1)
```

5.
```
program TESTPRIME;
var
    NUMBER, COUNT : INTEGER;
    PRIME : BOOLEAN;
begin
    WRITELN('Какое число следует проверить?');
    READ(NUMBER);
    COUNT := 2;
    PRIME := TRUE;
```

```
        while COUNT > NUMBER do
            begin
                if (NUMBER mod COUNT) <> 0 then
                    PRIME := FALSE;
                    COUNT := COUNT + 1
            end;
        if PRIME then
            WRITELN(NUMBER : 5, ' простое число.')
        else
            WRITELN(NUMBER : 5, ' не простое число.')
end.
```

**Глава 7**

1. Заметьте, что использован оператор *for... downto*, а не *for... to*. Так как начальное значение меньше конечного, действия вообще не будут выполнены.

4.
```
program USEFOR;
(* Эта программа читает список чисел и печатает их
   среднее. *)
const
      LISTSIZE = 10;
var
      SUM : REAL; (* посчитанная сумма *)
      NUMBER : REAL; (* введенное число *)
      COUNT : INTEGER; (* индекс цикла*)
begin
      SUM := 0;
      WRITELN('Пожалуйста, введите ваш ', LISTSIZE,
      ' — элементный список. ');
      for COUNT := 1 to LISTSIZE do
      begin
                READ(NUMBER);
                SUM := SUM + NUMBER
          end;
      WRITELN('Среднее значение равно ',
      SUM / LISTSIZE)
end.
```

5. Тридцать два раза.

**Глава 8**

1. Второй и третий операторы присваивания неверны, поскольку в них делается попытка присвоить значение, тип которого не соответствует типу переменной.

(Кроме того, четвертый оператор присваивания вызовет ошибку при SUNSIGN = CAPRICORN.)

2. Первое описание неверное, поскольку его нижняя граница (FRIDAY) больше верхней (MONDAY).

## Глава 9

1. array [BOOLEAN] of REAL
   array [HUES] of INTEGER
   array [1 .. 10] of 1 .. 10
2. Первое описание неверно из-за неправильного указания типа компонент, поскольку интервал не может иметь в качестве базового типа REAL. Второе неверно из-за неправильного указания типа индекса. Третье и четвертое описания правильны. Массив SALES содержит один элемент: SALES [10, 3].

## Глава 10

1.
```
Program CONVERT;
(* Эта программа вводит строку и заменяет в ней
   все литеры нижнего регистра на литеры верхнего
   регистра.*)
  const
        UPPER = 'ABCDEFGHIJKLMNOPQRSTUVW
        XYZ';
        LOWER = 'abcdefghijklmnopqrstuvwxyz';
var
      LINE STRING;
      COUNT, LETTER : INTEGER;
begin
      WRITELN('Введите, пожалуйста, строку для
                                  преобразования ')
      WRITELN(' ее литер к верхнему регистру. ');
      READLN(LINE);
      for COUNT := 1 to LENGTH(LINE) do
          if (LINE[COUNT] > = 'a') and
          (LINE[COUNT] < = 'z') then
          for LETTER := 1 to 26 do
            if LINE[COUNT] = LOWER[LETTER] then
              LINE[COUNT] = UPPER[LETTER];
      WRITELN(' Преобразованная строка: ');
      WRITELN(LINE)
end.
```

3. Да, верно. Значение строки безразлично для ЭВМ. Реакция на строку 'BEGIN' точно такая же, как, например, на строку 'IJKLM'.
4. Это выражение не имеет значения. ORD воспринимает аргументы типа CHAR, а не типа STRING.

6.
```
NEW_NAME := '';
for COUNT := 1 to SIZE do
    NEW_NAME[COUNT] := OLD_NAME[COUNT
                              + START - 1];
```

7. в) Попытка модифицировать константу,
   г) Задает STUDENT_NAME длину, превышающую атрибут длины переменной.

## Глава 11

1. Третья функция неверна, так как const — зарезервированное слово. Четвертая хотя и верна, но нежелательна, так как она придает новый смысл задействованному идентификатору MAXINT.
2. VALUE
3.
```
function ISDIGIT(ONECHAR : CHAR) : BOOLEAN;
begin
    ISDIGIT := (ONECHAR >= '0') and
    (ONECHAR <= '9')
end; (* функции ISDIGIT *)
```

## Глава 12

1.
```
procedure SWAP(var FIRST, SECOND : REAL);
var TEMP : REAL;
begin
    TEMP := FIRST;
    FIRST := SECOND;
    SECOND := TEMP
end; (* процедуры SWAP *)
```
4.
```
type
    COEFF = (A, B, C);
    POLYNOMIAL = array [COEFF] of INTEGER;
procedure QUADMULTIPLY(FIRST, SECOND :
                                POLYNOMIAL;
                       var RESULT : POLYNOMIAL);
var INDEX : COEFF;
begin
```

```
    for INDEX := A to C do
        RESULT[INDEX] := 0;
    if (FIRST[A] <> 0) or (SECOND[A] <> 0) then
        WRITELN('Заданный полином — не линей-
                                        ный.')
    else
        begin
            RESULT[A] := FIRST[B] * SECOND[B];
            RESULT[B] := FIRST[B] *
            SECOND[C] + FIRST[C] * SECOND[B];
            RESULT[C] := FIRST[C] * SECOND[C]
        end
end; (* процедуры QUADMULTIPLY *)
```

**Глава 14**

1. Неверны только а) и г). Если вам не понравились в) и д), вспомните, что CHAR и BOOLEAN (и их интервалы) являются порядковыми типами.
2. а), б), г) и е) верны.

**Глава 15**

1. а) INTEGERGER
   б) REMINDER
   в) (неверно)
   г) DATE
   д) STRING
   е) CHAR
   ж) (неверно)
   з) DATE
3. е) и ж) невозможны, так как тип множества должен быть порядковым.

**Глава 16**

1.
```
program TEXTCOUNT;
var
    TEXTFILE : TEXT;
    CHAR_COUNT, LINE_COUNT : INTEGER;
    FILE_NAME : STRING;
    ONE_LINE : STRING;
begin
    WRITELN('Введите, пожалуйста, имя внешнего
                                        файла.');
    WRITELN('строки и литеры которого нужно со-
                                        считать.');
```

```
    READLN(FILE_NAME);
    RESET(TEXTFILE, FILE_NAME);
    CHAR_COUNT := 0;
    LINE_COUNT := 0;
    while not EOF(TEXTFILE) do
        begin
            READLN(TEXTFILE, ONE_LINE);
            CHAR_COUNT := CHAR_COUNT +
                          + LENGTH(ONE_LINE);
            LINE_COUNT := LINE_COUNT + 1
        end;
    WRITELN('Прочитано ', CHAR_COUNT,' литер.');
    WRITELN('Прочитано ', LINE_COUNT,' строк.');
end.
```

**4.**
```
program ENCODE2;
  var
      KEY_PRESSED : CHAR;
  begin
      while not EOF(KEYBOARD) do
          begin
              while not EOLN(KEYBOARD) do
                  begin
                      READ(KEYBOARD,
                      KEY_PRESSED);
                      if ORD(KEY_PRESSED)
                      = 255 then
                          KEY_PRESSED := CHR(0)
                      else
                          KEY_PRESSED := SUCC
                          (KEY_PRESSED);
                      WRITE(KEY_PRESSED)
                  end;
              WRITELN
          end
  end.
```

**6.**
```
procedure KILL_FILE(FILE_ID : STRING);
  var
      TEMP_FILE : TEXT;
  begin
      REWRITE(TEMP_FILE, FILE_ID);
      CLOSE(TEMP_FILE, PURGE)
  end: (* процедуры KILL_FILE *)
```

```
4. function FINDNODE(BASE : TREELINK;
                     NODE : STRING) : BOOLEAN;
   begin
        if BASE = NIL then
            FINDNODE := FALSE
        else
            with BASE↑ do
                if DATA = NODE then
                    FINDNODE := TRUE
                else
                    FINDNODE := FINDNODE(LEFT,
                                NODE) or FINDNODE
                                (RIGHT, NODE)
   end; (* функции FINDNODE *)

6. function EQUALTREE(BASE1, BASE2 : TREELINK) :
                                          BOOLEAN;
   begin
        EQUALTREE := FALSE;
        if BASE1 = BASE2 then
            EQUALTREE := TRUE
        else
            if (BASE1 <> NIL)  and  (BASE2 <> NIL)
                                                then
                if BASE1↑.DATA = BASE2↑.DATA
                                                then
                    EQUALTREE := EQUALTREE
                        (BASE1↑.LEFT,
                        BASE2↑.LEFT) and
                        EQUALTREE(BASE1↑
                            .RIGHT, BASE2↑.RIGHT)
   end; (* функции EQUALTREE *)

8. procedure POSTORDER(BASE : TREELINK);
   begin
        if BASE <> NIL then
            begin
                POSTORDER(BASE↑.LEFT);
                POSTORDER(BASE↑.RIGHT);
                WRITELN(BASE↑.DATA)
            end
   end; (* процедуры POSTORDER *)
```

## Глава 18

1. Этот цикл пропускает литеры из INPUT, пока не встретится знак вопроса.
2. Варианты г) и е) невозможны, поскольку файлы не могут быть образованы в пределах другого структурированного типа. Вариант е) невозможен еще и потому, что тип компонент множества должен быть порядковым типом.

# Предметный указатель


Деревья двоичные 191–195

Заголовок 10, 103–105, 122, 126
Запись 151–154
– с вариантами 157–161, 186–187
– упакованная 161
Зарезервированные слова 11, 216

Имя переменной 11–12

Код ASCII 88, 217–218
Комментарий 52–53
Константа 35–37, 70, 79, 94–95
– MAXINT 37, 218

Массив 76–83
– литер 90–92
– упакованный 83–84
Метка, переход 63
Множество 140–148
– ограничение на размер 144
– операции 144–147
– тип компонент 140–141

Несоответствие типов 30

Оператор присваивания 14, 18, 80, 86
– and 33–34
– div 13, 27
– goto 63–64, 127
– if... then... else 39–41, 60
– in 142
– mod 13–14, 34–35
– not 33–34
– or 33–34
– with 154–157
Описание типа 71–72
Отладка 18, 137–138
Отношения 32
Очередь 178–183

Параметр-переменная 125–127
Переменная файловая KEYBOAD 170
Правила локализации 108–114, 127
Примеры контрольные 135
Процедура встроенная 216–217
– DELETE 97–98
– EXIT 128
– GET 198–202
– GOTOXY 219–220
– INSERT 98
– NEW 184–186
– PAGE 173–179
– PUT 199–202
– READ и READLN 22–24, 84, 91, 93, 165, 169–171
– RESET и REWRITE 164–168, 172–173, 197–200
– SEEK 202–203
– STR 98–99
– WRITE и WRITELN 16, 20–22, 91, 93, 165, 168–171
Порядок описания 123

Размер поля 21, 29
Рекурсия 115–120, 124, 128

Сообщение побуждающее 50
Список связный 187–190, 207–214
Ссылка вперед 118–120
Стек 178–181
Строка 21, 90–100

Тип данных BOOLEAN 32–35
– – CHAR 86–90, 200–201
– – INTEGER 11–14, 30–32
– – REAL 27–30
– – STRING 92–100
– индексов массива 76–77, 82, 88
– интервальный 69–71
– перечисляемый 66–69
– порядковый 72–74
– простой 76
Точка с запятой 17, 41–43

Указатель 183–187
– значение NIL 185

Файл 164–177
– атрибуты по умолчанию 169
– буферная переменная 197–198
– внешний 166
– имена устройств 167
– режим закрытия 174–177
– строки логические 170–171
– тип INPUT 165
– – INTERACTIVE 165
– – OUTPUT 165
– – TEXT 165
Форма представления вещественных чисел 28–29
– экспоненциальная 28
Функция ABS 13, 27
– ANAN 28
– ARCTAN 28
– CHR 88
– CONCAT 96
– COPY 97
– COS 27
– EOF 45–46, 170, 173, 198–199
– EOLN 171–173
– EXP 2–7
– FORESULT 168
– LENGTH 93
– LN 27
– LOG 27
– ODD 34–35
– ORD 74, 88
– POS 96
– PRED 72–74, 88
– ROUND 31
– SIN 27
– SQR 13, 27
– SQRT 27, 30–32
– SUCC 72–74, 88
– TRUNC 31

Целые с увеличенной точностью 218–219
Цикл for 56–60, 82–83, 88
– repeat 54–56
– while 43–46, 54, 56

# Оглавление

Предисловие редактора перевода . . . . . 5
**Глава 1. Введение** . . . . . 8

**Глава 2. Простая программа** . . . . . 10

2.1. Как выглядит программа? . . . . . 10
2.2. Описание переменных . . . . . 11
2.3. Тело программы . . . . . 12
2.4. Для чего служит точка с запятой? . . . . . 17
2.5. Синтаксические ошибки . . . . . 17

**Глава 3. Ввод и вывод информации** . . . . . 20

3.1. Операторы WRITE и WRITELN . . . . . 20
3.2. READ и READLN . . . . . 22
3.3. Проектирование вывода . . . . . 24

**Глава 4. Встроенные типы данных** . . . . . 27

4.1. Тип данных REAL (вещественный) . . . . . 27
4.2. Взаимные преобразования чисел . . . . . 30
4.3. Тип данных BOOLEAN . . . . . 32
4.4. Описание констант . . . . . 35

**Глава 5. Условный оператор** . . . . . 39

5.1. Ветвление . . . . . 39
5.2. Составной оператор . . . . . 41
5.3. Организация циклов . . . . . 43

**Глава 6. Являетесь ли вы хорошим программистом?** . . . . 48

6.1. На кого следует ориентироваться при разработке программы . . . . . 48
6.2. Что рекомендуется и что не рекомендуется хорошим программистам . . . . . 49
6.3. Использование комментариев для пояснений . . . . 52

**Глава 7. Другие условные операторы** . . . . . 54

7.1. Еще о построении циклов . . . . . 54
7.2. Циклы со счетчиком . . . . . 56
7.3. Ветвление по ряду условий . . . . . 60
7.4. Безусловные переходы . . . . . 63

**Глава 8. Типы данных, определяемые программистом** . . . . 66

8.1. Перечисляемый тип . . . . . 66
8.2. Интервальный тип . . . . . 69
8.3. Описания типов . . . . . 71
8.4. Свойства порядковых типов . . . . . 72

**Глава 9. Массивы** . . . . . 76

9.1. Одномерные массивы . . . . . 76
9.2. Некоторые типичные операции над массивами . . . 79
9.3. Многомерные массивы . . . . . 81
9.4. Упакованные массивы . . . . . 83

**Глава 10. Типы данных, представляющие слова** . . . . . 86

10.1. Тип данных CHAR . . . . . 86
10.2. Массивы литер . . . . . 90
10.3. Тип данных STRING . . . . . 92
10.4. Встроенные операции над строками . . . . . 96

**Глава 11. Описание функций, определяемых программистом** . . 102

11.1. Описание функций . . . . . 102
11.2. Локальные и глобальные идентификаторы . . . . . 107
11.3. Остерегайтесь побочных эффектов . . . . . 114
11.4. Рекурсивные функции . . . . . 115

**Глава 12. Описание процедур, определяемых программистом** 122
12.1. Описание процедур . . . . . 122
12.2. Параметры-переменные . . . . . 125
12.3. Еще раз о передаче управления . . . . . 127
12.4. Преимущества модульности . . . . . 128

**Глава 13. Важность надежности программ** . . . . . 133

13.1. Почему программы допускают ошибки? . . . . . 133
13.2. Тестирование . . . . . 135
13.3. Отладка . . . . . 137
13.4. Подпрограммы и изоляция ошибок . . . . . 138

**Глава 14. Множества** . . . . . 140

14.1. Описание множеств . . . . . 140
14.2. Действия над множествами . . . . . 144
14.3. Сравнение множеств . . . . . 147

**Глава 15. Записи** . . . . . 151

15.1. Переменные типа RECORD . . . . . 151
15.2. Оператор WITH . . . . . 154
15.3. Записи с вариантами . . . . . 157
15.4. Упакованные записи . . . . . 161

**Глава 16. Текстовые файлы** . . . . . 164

16.1. Описание текстовых файлов . . . . . 164
16.2. Открытие текстовых файлов . . . . . 165
16.3. Операции над текстовыми файлами . . . . . 168
16.4. Закрытие текстовых файлов . . . . . 174

**Глава 17. Динамические структуры данных** . 178

17.1. Стеки и очереди . . . . . 178
17.2. Переменные-указатели . . . . . 183
17.3. Связный список . . . . . 187
17.4. Бинарные деревья . . . . . 191

**Глава 18. Файлы** . . . . . 197

18.1. Обработка последовательных файлов . . . . . 197
18.2. Режим произвольного доступа . . . . . 202
18.3. Связные списки в файлах . . . . . 207

**Приложения** . . . . . 216

Приложение А. Зарезервированные слова . . . . . 216
Приложение Б. Стандартные идентификаторы . . . . . 216
Приложение В. Код ASCII . . . . . 217
Приложение Г. Целые с увеличенной точностью . . . 218
Приложение Д. Управление курсором с помощью GOTOXY 219
Приложение Е. Ответы на вопросы, отмеченные '*' . . 220

Предметный указатель . . . . . 229

Практическое руководство

Давид Прайс

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ПАСКАЛЬ

Старший научный редактор Т. Н. Шестакова
Младший научный редактор Н. И. Сивилева
Художник А. И. Чаузов
Художественный редактор И. М. Иванов
Технический редактор Л. П. Бирюкова
Корректор Т. П. Пашковская

ИБ № 5783

Сдано в набор 15.04.86. Подписано к печати 3.09.86. Формат 84 × 108¹/₃₂. Бумага типографская кн.-журн. сыкт. Печать высокая. Гарнитура литературная. Объем 3,63 бум. л. Усл. печ. л. 12,18. Усл. кр.-отт. 12,41. Уч. изд. л. 11,17. Изд. № 6/4625. Тираж 75 000 экз. Зак. 1263. Цена 85 коп.

ИЗДАТЕЛЬСТВО «МИР» 129820, ГСП, Москва, И-110, 1-й Рижский пер., 2

Ярославский полиграфкомбинат Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли.
150014, Ярославль, ул. Свободы, 97

85 к.




Издательство
«Мир»