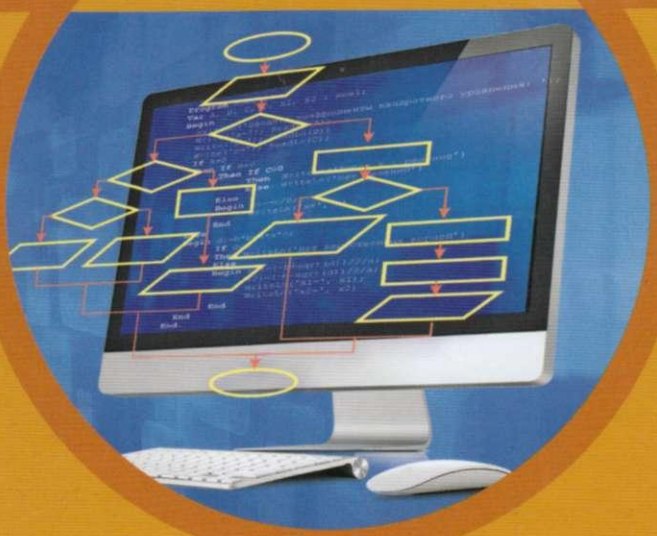
И. Г. Семакин, А. П. Шестаков

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Учебник



Профессиональное образование

ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ACADEMIA

И.Г.СЕМАКИН, А.П.ШЕСТАКОВ

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

УЧЕБНИК

Рекомендовано
Федеральным государственным автономным учреждением
«Федеральный институт развития образования»
в качестве учебника для использования в учебном процессе
образовательных учреждений, реализующих программы
среднего профессионального образования по специальностям
«Компьютерные системы и комплексы», «Информационные системы
(по отраслям)», учебная дисциплина «Основы алгоритмизации
и программирования»

Регистрационный номер рецензии 308
от 25 июня 2012 г. ФГАУ «ФИРО»

3-е издание, стереотипное

ACADEMIA

Москва
Издательский центр «Академия»
2016

УДК 681.3.06(075.32)
ББК 22.18я723
С30

Рецензенты:
канд. физ.-мат. наук, преподаватель ГАОУ СПО «Колледж предпринимательства № 11» *Е. И. Ночка*;
старший научный сотрудник ГНУ ГОСНИТИ Россельхозакадемии *А. А. Соломашкин*

Семакин И. Г.

С30 Основы алгоритмизации и программирования : учебник для студ. учреждений сред. проф. образования / И. Г. Семакин, А. П. Шестаков. — 3-е изд., стер. — М. : Издательский центр «Академия», 2016. — 304 с.

ISBN 978-5-4468-3155-5

Учебник создан в соответствии с требованиями Федерального государственного образовательного стандарта среднего профессионального образования по специальностям «Программирование в компьютерных системах»; «Компьютерные системы и комплексы» и «Информационные системы (по отраслям)»; ОП «Основы алгоритмизации и программирования».

Рассмотрены основы принципы алгоритмизации и программирования на базе языка Паскаль (версия Турбо Паскаль 7.0). Даны основные понятия объектно-ориентированного программирования и его реализация на языке Турбо Паскаль. Описана интегрированная среда программирования Delphi и визуальная технология создания графического интерфейса программ. Показана разработка программных модулей в этой среде.

Для студентов учреждений среднего профессионального образования.

УДК 681.3.06(075.32)
ББК 22.18я723

ISBN 978-5-4468-3155-5

Уважаемый читатель!

Данный учебник является частью учебно-методического комплекта по специальностям «Программирование в компьютерных системах», «Компьютерные системы и комплексы» и «Информационные системы (по отраслям)» и предназначен для изучения общепрофессиональной дисциплины «Основы алгоритмизации и программирования».

Учебник предназначен для изучения общепрофессиональной дисциплины «Основы алгоритмизации и программирования».

Учебно-методические комплекты нового поколения включают в себя традиционные и инновационные учебные материалы, позволяющие обеспечить изучение общеобразовательных и общепрофессиональных дисциплин и профессиональных модулей. Каждый комплект содержит учебники и учебные пособия, средства обучения и контроля, необходимые для освоения общих и профессиональных компетенций, в том числе и с учетом требований работодателя.

Учебные издания дополняются электронными образовательными ресурсами. Электронные ресурсы содержат теоретические и практические модули с интерактивными упражнениями и тренажерами, мультимедийные объекты, ссылки на дополнительные материалы и ресурсы в Интернете. В них включен терминологический словарь и электронный журнал, в котором фиксируются основные параметры учебного процесса: время работы, результат выполнения контрольных и практических заданий. Электронные ресурсы легко встраиваются в учебный процесс и могут быть адаптированы к различным учебным программам.

Предисловие

Программирование все в большей степени становится занятием лишь для профессионалов. Объявленный в середине 1980-х гг. лозунг «Программирование — вторая грамотность» остался в прошлом. В понятие «компьютерная грамотность» сегодня входит, прежде всего, навык использования многообразных средств информационных технологий. При решении той или иной информационной задачи сначала следует попытаться подобрать адекватное программное средство (электронные таблицы, системы управления базами данных, математические пакеты и др.), и только если эти средства не позволяют решить поставленную задачу, использовать универсальные языки программирования.

Различают программистов двух категорий: системных и прикладных. Системные программисты — это разработчики базовых программных средств ЭВМ (операционных систем, трансляторов, сервисных средств и т.д.), являющиеся профессионалами высочайшего уровня. Прикладные программисты разрабатывают средства программного обеспечения ЭВМ, предназначенные для решения задач в отдельных областях деятельности (науке, технике, производстве, сфере обслуживания, обучении и т.д.). Требования к качеству прикладных программ так же высоки, как и к качеству системных. Любая программа должна не только правильно решать задачу, но и иметь современный интерфейс, быть высоконадежной, дружественной к пользователю и т.д. Только такие программы могут выдержать конкуренцию на мировом рынке программных продуктов.

По мере развития компьютерной техники развивались методика и технология программирования. Сначала возникли командное и операторное программирование, в 1960-х гг. бурно развивались структурное программирование, линии логического и функционального программирования, а в настоящее время широко распространяются объектно-ориентированное и визуальное программирование.

Задача, которую следует ставить при первоначальном изучении программирования, — это освоение основ его структурной мето-

дики. Структурная методика до настоящего времени остается основой программистской культуры. Не освоив ее, человек, взявшийся изучать программирование, не имеет никаких шансов стать профессионалом.

В 1969 г. швейцарским профессором Никлаусом Виртом был создан язык программирования Паскаль, предназначавшийся для обучения студентов структурной методике программирования. Язык получил свое название в честь Блеза Паскаля — изобретателя первого вычислительного механического устройства. Позднее фирма Borland International, Inc (США) разработала систему программирования Турбо Паскаль для персональных компьютеров, которая вышла за рамки учебного применения и стала использоваться для научных и производственных целей. В Турбо Паскаль были внесены значительные дополнения к базовому стандарту Паскаля, описанного Н. Виртом. Со временем язык развивался. Начиная с версии 5.5, в Турбо Паскаль вводятся средства поддержки объектно-ориентированного программирования. В дальнейшем это привело к созданию Object Pascal — языка с возможностями объектно-ориентированного программирования. В начале 1990-х гг. объединение элементов объектно-ориентированного программирования в Object Pascal с визуальной технологией программирования привело к созданию системы программирования Delphi.

В *главе 1* настоящего учебника рассматриваются базовые понятия, относящиеся к любому процедурному языку программирования высокого уровня. Основное внимание уделяется структурной методике построения алгоритмов: использованию базовых алгоритмических структур, выделению в решаемой задаче подзадач и составлению вспомогательных алгоритмов. На этой основе можно переходить к изучению любого процедурного языка, поддерживающего структурную методику.

В *главе 2* дается описание языка Паскаль в варианте Турбо Паскаль (версия 7.0). В языках Object Pascal и Delphi сохраняется преемственность к Турбо Паскалю.

Содержание *главы 3* ориентировано на глубокое освоение учащимися базовых понятий языков программирования высокого уровня на примере Паскаля. Такая подготовка облегчает изучение других языков программирования в будущем.

В *главе 4* излагаются основы объектно-ориентированного программирования на примере их реализации в Object Pascal. Здесь же рассматривается язык программирования Delphi, являющийся объектно-ориентированным расширением языка Паскаль, с реализацией технологии визуального программирования.

При подготовке к изучению данного курса желательно усвоение учащимися основ алгоритмизации в рамках школьного базового курса информатики. Обычно в школе алгоритмизация изучается с использованием учебных исполнителей, позволяющих успешно освоить основы структурной методики:

- построение алгоритмов из базовых структур;
- применение метода последовательной детализации.

Желательно также иметь представление об архитектуре ЭВМ на уровне машинных команд (достаточно на модельных примерах учебных компьютеров, изучаемых в школьной информатике, т.е. не обязательно знание реальных языков команд или ассемблера). Это позволяет усвоить основные понятия программирования (переменной, присваивания); «входить в положение транслятора» и не делать ошибок, даже не помня каких-то деталей синтаксиса языка; предвидеть те «подводные камни», на которые может «напороться» программа в процессе выполнения. По существу все эти качества и отличают профессионального программиста от дилетанта.

Еще одно качество профессионала — это способность воспринимать красоту программы, т.е. получать эстетическое удовольствие от хорошо написанной программы. Нередко это чувство помогает интуитивно отличить неправильную программу от правильной. Однако основным критерием оценки программы должна быть, безусловно, не интуиция, а грамотно организованное тестирование.

Процесс изучения и практического освоения программирования подразделяется на три части:

- изучение методов построения алгоритмов;
- изучение языка программирования;
- изучение и практическое освоение определенной системы программирования.

Решению задач первой части посвящены главы 1 и 3 данного учебника. В главе 1 даются основные, базовые, понятия и принципы построения алгоритмов работы с величинами. В главе 3 излагаются некоторые известные методики полного построения алгоритмов, рассматриваются проблемы тестирования программ и оценки сложности алгоритмов.

Языки программирования Турбо Паскаль и Delphi излагаются соответственно в главах 2 и 4 учебника. Однако подчеркнем, что данная книга — это, прежде всего, учебник по программированию, а не по языкам Паскаль и Delphi, поэтому исчерпывающего описания данных языков вы здесь не найдете, они излагаются в

объеме, необходимом для начального курса программирования. Более подробное описание этих языков можно найти в книгах, указанных в списке литературы.

В данном учебнике нет инструкций по работе с конкретными системами программирования для изучаемых языков, с ними студенты должны ознакомиться в процессе выполнения практических работ на ЭВМ, используя специальную литературу.

Глава 1

ОСНОВНЫЕ ПРИНЦИПЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Из данной главы вы узнаете:

- последовательность этапов решения задачи на компьютере средствами программирования;
- что такое алгоритм решения задачи на компьютере;
- с какими данными работает компьютер;
- с помощью какого набора команд (системы команд) можно построить любой алгоритм решения задачи на компьютере;
- правила описания алгоритмов на блок-схемах и на учебном алгоритмическом языке;
- три основные алгоритмические структуры — следование, ветвление, цикл;
- что такое трассировка алгоритма;
- логические основы программирования; сущность понятий «логическая величина», «логическое выражение», «логические операции»;
- правила выполнения логических операций и вычисления логических выражений;
- что такое вспомогательный алгоритм и как он описывается на алгоритмическом языке в форме процедуры;
- как происходит обращение к процедуре в основном алгоритме;
- в чем состоят принципы структурного программирования;
- историю развития языков и технологий программирования;
- классификацию языков программирования;

- что такое трансляция и какие существуют способы трансляции;
- состав и структуру процедурных языков программирования высокого уровня;
- способы описания синтаксиса языков программирования.

Вы научитесь:

- описывать алгоритмы различной структуры (линейные, ветвящиеся, циклические) в виде блок-схем и на учебном алгоритмическом языке;
- записывать логические выражения в качестве условий в ветвлениях и циклах;
- выполнять трассировку алгоритмов;
- описывать процедуры на алгоритмическом языке и обращения к ним из основного алгоритма.

1.1. АЛГОРИТМЫ И ВЕЛИЧИНЫ

Этапы решения задачи на ЭВМ. Работа по решению любой задачи с использованием компьютера включает в себя следующие шесть этапов:

1. Постановка задачи.
2. Формализация задачи.
3. Построение алгоритма.
4. Составление программы на языке программирования.
5. Отладка и тестирование программы.
6. Проведение расчетов и анализ полученных результатов.

Часто эту последовательность называют *технологической цепочкой решения задачи на ЭВМ* (непосредственно к программированию из этого списка относятся п. 3...5).

На этапе постановки задачи следует четко определить, *что дано* и *что требуется найти*. Важно описать полный набор исходных данных, необходимых для решения задачи.

На этапе формализации чаще всего задача переводится на язык математических формул, уравнений и отношений. Если решение задачи требует математического описания какого-то реального объекта, явления или процесса, то ее формализация равносильна получению соответствующей математической модели.

Третий этап — это построение алгоритма. Опытные программисты часто сразу пишут программы на определенном языке, не прибегая к каким-либо специальным средствам описания алгоритмов (блок-схемам, псевдокодам), однако в учебных целях полезно сначала использовать эти средства, а затем переводить полученный алгоритм на язык программирования.

Первые три этапа — это работа без компьютера. Последующие два этапа — это собственно программирование на определенном языке в определенной системе программирования. На последнем — шестом — этапе разработанная программа уже используется в практических целях.

Таким образом, программист должен уметь строить алгоритмы, знать языки программирования, уметь работать в соответствующей системе программирования.

Основой профессиональной грамотности программиста является развитое алгоритмическое мышление.

Понятие алгоритма. Одним из фундаментальных понятий в информатике является понятие алгоритма. Сам термин «алгоритм», заимствованный из математики, происходит от *лат.* Algorithmi — написание имени Мухамеда аль-Хорезми (787—850), выдающегося математика средневекового Востока. В XII в. был осуществлен латинский перевод его математического трактата, из которого европейцы узнали о десятичной позиционной системе счисления и правилах арифметики многозначных чисел. Именно эти правила в то время называли алгоритмами. Сложение, вычитание, умножение «столбиком», деление «уголком» многозначных чисел — это первые алгоритмы в математике. Правила алгебраических преобразований и вычисление корней уравнений также можно отнести к математическим алгоритмам.

В наше время понятие алгоритма трактуется шире. *Алгоритм* — это последовательность команд управления каким-либо исполнителем. В школьном курсе информатики с понятием алгоритма и методами построения алгоритмов ученики знакомятся на примерах учебных исполнителей: Робота, Черепахи, Чертежника и др. Эти исполнители ничего не вычисляют. Они создают рисунки на экране, перемещаются в лабиринтах, перетаскивают предметы с места на место. Таких исполнителей принято называть *исполнителями, работающими в обстановке.*

В разделе «Программирование» информатики изучаются методы программного управления работой ЭВМ, т. е. в качестве исполнителя выступает компьютер. Компьютер работает с величинами — различными информационными объектами: числами, сим-

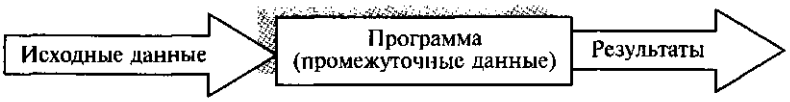


Рис. 1.1. Уровни данных относительно программы

волами, кодами и др., поэтому алгоритмы, предназначенные для управления компьютером, называются *алгоритмами работы с величинами*.

Данные и величины. Совокупность величин, с которыми работает компьютер, принято называть *данными*. По отношению к программе различают *исходные*, *окончательные* (*результаты*) и *промежуточные* данные, которые получают в процессе вычислений (рис. 1.1).

Например, при решении квадратного уравнения $ax^2 + bx + c = 0$ исходными данными являются коэффициенты a, b, c, результатами — корни уравнения x_1, x_2, а промежуточными данными — дискриминант уравнения $D = b^2 - 4ac$.

Для успешного освоения программирования необходимо усвоить следующее правило: *всякая величина занимает свое определенное место в памяти ЭВМ*, иногда говорят — ячейку памяти. Термин «ячейка» для архитектуры современных ЭВМ несколько устарел, однако в учебных целях его удобно использовать.

Любая величина имеет три основных свойства: *имя*, *значение* и *тип*. На уровне команд процессора величина идентифицируется адресом ячейки памяти, в которой она хранится. В алгоритмах и языках программирования величины подразделяются на константы и переменные. *Константа* — неизменная величина, и в алгоритме она представляется собственным значением, например: 15, 34.7, *k*, True и др. *Переменные величины* могут изменять свои значения в ходе выполнения программы и представляются в алгоритме символическими именами — идентификаторами, например: X, S2, cod15 и др. Любые константы и переменные занимают ячейку памяти, а значения этих величин определяются двоичным кодом в этой ячейке.

Теперь о типах величин — *типах данных* — понятии, которое встречается при изучении в курсе информатики баз данных и электронных таблиц. Это понятие является фундаментальным в программировании.

В каждом языке программирования существует своя концепция и своя система типов данных. Однако в любой язык входит

минимально необходимый набор основных типов данных: *целые*, *вещественные*, *логические* и *символьные*. Типы величин характеризуются множеством допустимых значений, множеством допустимых операций, формой внутреннего представления (табл. 1.1).

Типы констант определяются по контексту (т.е. по форме записи в тексте), а типы переменных устанавливаются в описаниях переменных.

По структуре данные подразделяются на *простые* и *структурированные*. Для простых величин, называемых также скалярными, справедливо утверждение *одна величина — одно значение*, а для структурированных — *одна величина — множество значений*.

Таблица 1.1. Основные типы данных

Тип	Значения	Операции	Внутреннее представление
Целые	Целые положительные и отрицательные числа в некотором диапазоне, например: 23, −12, 387	Арифметические операции с целыми числами: сложение, вычитание, умножение, целое деление и деление с остатком. Операции отношений (<, >, = и др.)	Формат с фиксированной точкой
Вещественные	Любые (целые и дробные) числа в некотором диапазоне, например: 2.5, −0.01, 45.0, 3.6×10^9	Арифметические операции. Операции отношений	Формат с плавающей точкой
Логические	True (истина) False (ложь)	Логические операции: И (and), ИЛИ (or), НЕТ (not). Операции отношений	1 — True; 0 — False
Символьные	Любые символы компьютерного алфавита, например: a, 5, +, $	Операции отношений	Коды таблицы символьной кодировки, например: ASCII — один символ — 1 байт; Unicode — один символ — 2 байт

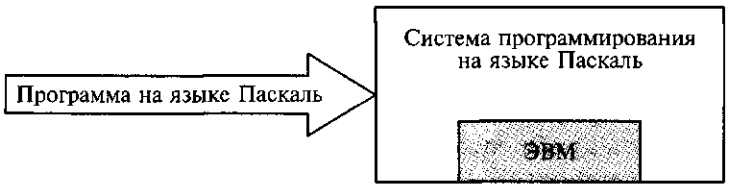


Рис. 1.2. Компьютер как исполнитель программ на языке Паскаль

К структурированным величинам относятся массивы, строки, множества и др.

ЭВМ — исполнитель алгоритмов. Как известно, каждый алгоритм (программа) составляется для конкретного исполнителя, т. е. в рамках его системы команд. О каком же исполнителе идет речь при изучении темы «Программирование для ЭВМ»? Ответ очевиден: исполнителем здесь является компьютер, а точнее говоря, комплекс ЭВМ + система программирования (СП). Программист составляет программу на том языке, на который ориентирована СП. Схематически это изображено на рис. 1.2, где *входным языком* исполнителя является язык программирования Паскаль.

Независимо от того, на каком языке программирования будет написана программа, алгоритм решения любой задачи на ЭВМ может быть составлен из следующих команд: присваивания, ввода, вывода, обращения к вспомогательному алгоритму, цикла, ветвления.

Далее для описания алгоритмов будут использоваться блок-схемы и учебный алгоритмический язык (АЯ), применяемый в школьном курсе информатики.

1.2. ЛИНЕЙНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ АЛГОРИТМЫ

Основным элементарным действием в вычислительных алгоритмах является *присваивание значения переменной величине.*

Если значение константы определено видом ее записи, то переменная величина получает конкретное значение только в результате присваивания, которое может осуществляться двумя способами: с помощью команды присваивания и с помощью команды ввода.

Рассмотрим пример. В школьном учебнике математики правило деления двух обыкновенных дробей описано следующим образом:

1) числитель первой дроби умножить на знаменатель второй дроби;

2) знаменатель первой дроби умножить на числитель второй дроби;

3) записать дробь, числитель которой есть результат выполнения п. 1, а знаменатель — результат выполнения п. 2.

Алгебраическая форма записи этого примера следующая:

$$\frac{a}{b} : \frac{c}{d} = \frac{a \cdot d}{b \cdot c} = \frac{m}{n}.$$

Теперь построим алгоритм деления дробей для ЭВМ, сохранив те же обозначения переменных, которые были использованы в алгебраическом выражении.

Исходными данными здесь являются целочисленные переменные *a*, *b*, *c*, *d*, а результатом — целые величины *m* и *n*. Блок-схема алгоритма деления дробей приведена на рис. 1.3. Данный алгоритм на учебном алгоритмическом языке будет иметь следующий вид:

```
алг деление дробей
нач
    цел a, b, c, d, m, n
    ввод a, b, c, d
    m := a × d
    n := b × c
    вывод m, n
кон
```

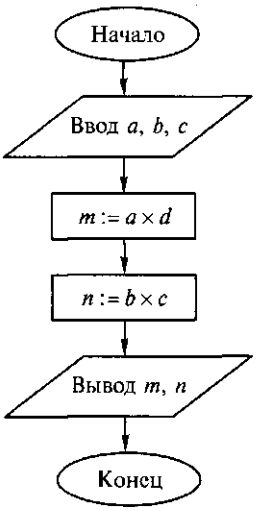


Рис. 1.3. Блок-схема алгоритма деления дробей

Формат команды присваивания следующий:

```
переменная := выражение
```

Знак «:=» следует читать как «присвоить».

Команда присваивания означает следующие действия, выполняемые компьютером:

1) вычисляется *выражение*;

2) полученное значение присваивается *переменной*.

В приведенном алгоритме присутствуют две команды присваивания. В блок-схемах команды присваивания изображаются в прямоугольниках. Такой блок называется *вычислительным*.

При описании алгоритмов не обязательно соблюдать строгие правила записи выражений, это можно делать в обычной математической форме, так как это еще не язык программирования со строгим синтаксисом.

В рассматриваемом алгоритме имеется команда ввода:

```
ввод a, b, c, d
```

В блок-схемах команда ввода записывается в параллелограмме — блоке ввода-вывода. При выполнении этой команды процессор прерывает работу и ожидает действий пользователя. Пользователь должен набрать на устройстве ввода (клавиатуре) значения вводимых переменных и нажать клавишу ввода <Enter>. Значения следует вводить в том же порядке, в каком эти переменные расположены в списке ввода. Обычно с помощью команды ввода присваиваются значения исходных данных, а команда присваивания используется для получения промежуточных и конечных величин.

Полученные компьютером результаты решения задачи должны быть сообщены пользователю, для чего и предназначена команда вывода:

```
вывод m, n
```

С помощью этой команды результаты выводятся на экран или через устройство печати на бумагу.

Поскольку присваивание является важнейшей операцией в вычислительных алгоритмах, обсудим ее более подробно.

Рассмотрим последовательное выполнение четырех команд присваивания, в которых участвуют две переменные величины — a, b. Для каждой команды присваивания в табл. 1.2 указаны значения переменных, которые устанавливаются после ее выполнения.

Этот пример иллюстрирует три основных свойства присваивания:

- пока переменной не присвоено значение, она остается неопределенной;
- значение, присвоенное переменной, сохраняется вплоть до выполнения следующего присваивания этой переменной;
- новое значение, присваиваемое переменной, заменяет ее предыдущее значение.

Таблица 1.2. Изменение значений переменных при выполнении команды присваивания

Команда	a	b
$a := 1$	1	—
$b := 2 \times a$	1	2
$a := b$	2	2
$b := a + b$	2	4

Таблица 1.3. Обмен значениями между переменными

Команда	X	Y	Z
ввод X, Y	1	2	—
$Z := X$	1	2	1
$X := Y$	2	2	1
$Y := Z$	2	1	1

Рассмотрим алгоритм, который часто используется при программировании. Даны две величины: X и Y. Требуется произвести между ними обмен значениями. Например, если сначала было $X = 1$, $Y = 2$, то после обмена должно стать $X = 2$, $Y = 1$.

Этой задаче аналогична следующая ситуация. Имеются два стакана: один — с молоком, другой — с водой. Требуется произвести между ними обмен содержимым. Ясно, что в этом случае необходим третий стакан — пустой. Последовательность действий при обмене будет такой:

1) перелить молоко из 1-го стакана в 3-й;
2) воду из 2-го стакана в 1-й;
3) молоко из 3-го стакана во 2-й.

Аналогично для обмена значениями двух переменных требуется третья дополнительная переменная. Назовем ее Z. Тогда задачу обмена значениями можно решить последовательным выполнением трех команд присваивания (табл. 1.3).

Пример со стаканами не совсем точно характеризует ситуацию обмена между переменными, так как при переливании жидкостей один из стаканов становится пустым. В результате же выполнения команды присваивания ($X := Y$) переменная, стоящая справа (Y), сохраняет свое значение.

Алгоритм деления дробей имеет линейную структуру, т. е. в нем все команды выполняются в строго определенной последовательности, каждая по одному разу. Линейный алгоритм состоит из команд присваивания, ввода, вывода и обращения к вспомогательным алгоритмам (что будет рассмотрено далее).

При описании алгоритмов с помощью блок-схем типы данных, как правило, не указываются (но подразумеваются). В учебных алгоритмах (на А Я) для всех переменных типы данных указываются

явно и их описание производится сразу после заголовка алгоритма. При этом используются следующие обозначения: **цел** — целые, **вещ** — вещественные, **лит** — символьные (литерные), **лог** — логические. В алгоритме деления дробей все переменные целого типа.

1.3. ВЕТВЛЕНИЯ И ЦИКЛЫ В ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМАХ

Составим алгоритм решения квадратного уравнения

$$ax^2 + bx + c = 0.$$

Эта задача хорошо знакома из математики. Исходными данными здесь являются коэффициенты a, b, c, а решением в общем случае — два корня (x_1 и x_2), которые вычисляются по формуле

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Все используемые в этой программе величины вещественного типа.

Алгоритм решения кваратного уравнения будет иметь следующий вид:

```
алг корни квадратного уравнения
вещ a, b, c, d, x1, x2
нач ввод a, b, c
    d  := b^2 - 4ac
    x1 := (-b + √d ) / (2a)
    x2 := (-b - √d ) / (2a)
    вывод x1, x2
кон
```

Недостаток такого алгоритма виден «невооруженным глазом». Он не обладает важнейшим свойством, предъявляемым к качественным алгоритмам: универсальностью по отношению к исходным данным. *Какими бы ни были значения исходных данных, алгоритм должен приводить к получению определенного результата и выполняться до конца.* Результатом может быть числовой ответ, но может быть и сообщение о том, что при таких данных задача решения не имеет. Недопустимы остановки в середине алгоритма

из-за невозможности выполнения какой-либо операции. Данное свойство в литературе по программированию называют *результативностью алгоритма* (получение какого-то результата в любом случае).

Для построения универсального алгоритма сначала требуется тщательно проанализировать математическое содержание задачи.

Решение квадратного уравнения зависит от значений коэффициентов a, b, c.

Проанализируем эту задачу, ограничиваясь поиском только вещественных корней:

если $a = 0$, $b = 0$, $c = 0$, любое значение x — решение уравнения;

если $a = 0$, $b = 0$, $c \neq 0$, уравнение решений не имеет;

если $a = 0$, $b \neq 0$, это линейное уравнение, имеющее одно решение $x = -c/b$;

если $a \neq 0$ и $d = b^2 - 4ac \geq 0$, уравнение имеет два вещественных корня x_1, x_2;

если $a \neq 0$ и $d < 0$, уравнение не имеет вещественных корней.

Блок-схема алгоритма решения квадратного уравнения показана на рис. 1.4.

Этот же алгоритм на алгоритмическом языке будет иметь следующий вид:

```
алг корни квадратного уравнения
вещ a, b, c, d, x1, x2
нач ввод a, b, c
    если  a = 0
    то    если  b = 0
          то    если  c = 0
                то вывод «Любое x — решение»
                иначе вывод «Нет решений»
                кв
          иначе x := -c/b
                вывод x
          кв
    иначе d := b^2 - 4ac
          если d < 0
          то вывод «Нет вещественных корней»
    иначе x1 := (-b + √d)/(2a); x2 := (-b - √d)/(2a)
          вывод «x1 =», x1, «x2 =», x2
    кв
кв
кон
```

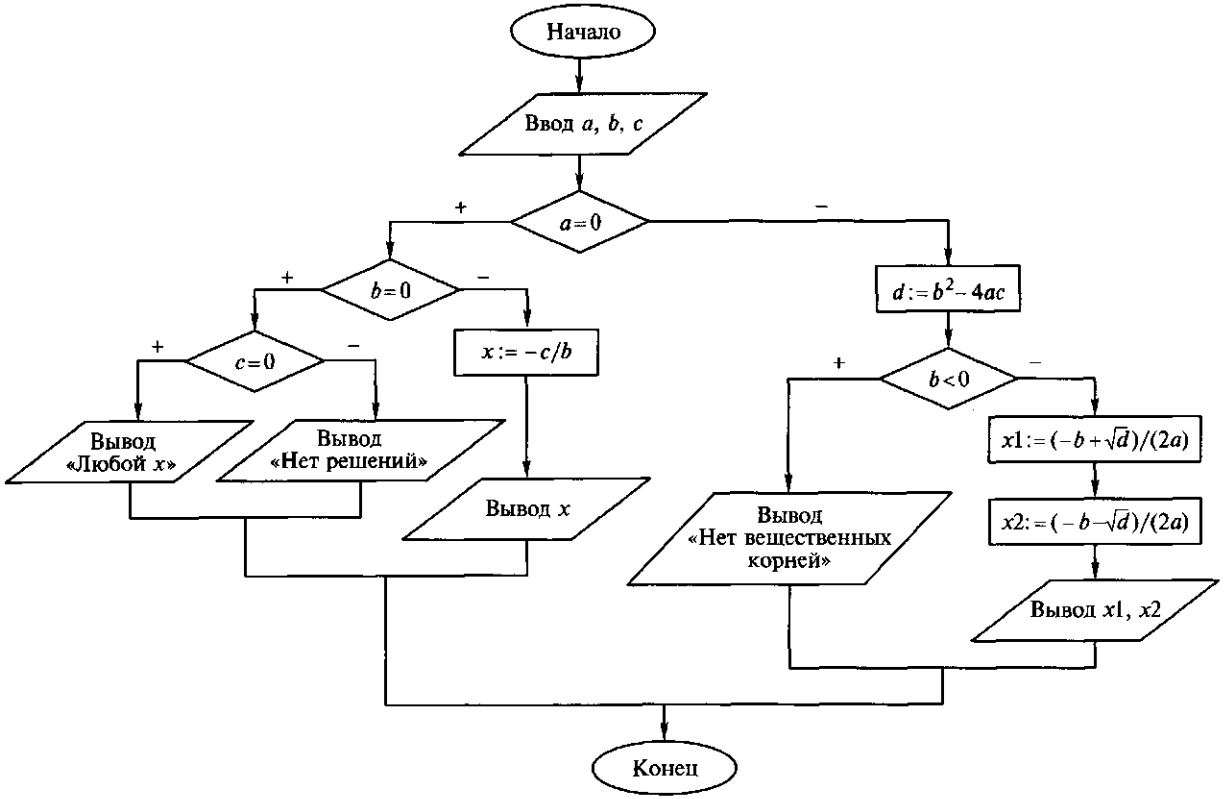


Рис. 1.4. Блок-схема алгоритма решения квадратного уравнения

В данном алгоритме многократно использована *структурная команда ветвления*, общий вид которой в виде блок-схемы показан на рис. 1.5. На алгоритмическом языке команду ветвления можно записать в следующем виде:

```
если условие
то серия 1
иначе серия 2
кв
```

В соответствии с приведенной блок-схемой команды ветвления сначала проверяется условие (вычисляется логическое выражение). Если условие истинно, то выполняется последовательность команд, на которую указывает стрелка с надписью «Да» (положительная ветвь) — «Серия 1». В противном случае выполняется отрицательная ветвь команд — «Серия 2».

На АЯ условие записывается после служебного слова «если», положительная ветвь — после слова «то», а отрицательная — после слова «иначе». Буквами «кв» в такой записи обозначают конец ветвления.

Если на ветвях одного ветвления содержатся другие ветвления, значит, алгоритм имеет структуру *вложенных ветвлений*. Именно такую структуру имеет алгоритм решения квадратного уравнения (см. рис. 1.4), в котором для краткости вместо слов «Да» и «Нет» использованы соответственно знаки «+» и «–».

Рассмотрим следующую задачу: дано целое положительное число n. Требуется вычислить $n!$ (n-факториал). Вспомним определение факториала:

$$n! = \begin{cases} 1, & \text{если } n = 0; \\ 1 \times 2 \times \ldots \times n, & \text{если } n \geq 1. \end{cases}$$

Блок-схема алгоритма вычисления $n!$ с тремя переменными целого типа: n — аргумент, i — промежуточная переменная, F — результат приведена на рис. 1.6. Для проверки правильности указанного алгоритма построим трассировочную табл. 1.4, в которой для конкретных значений исходных данных по шагам прослеживается изменение переменных, входящих в алгоритм. Данная таблица составлена для случая $n = 3$.

Приведенная трассировка доказывает правильность рассматриваемого алгоритма. Теперь запишем этот алгоритм на алгоритмическом языке:

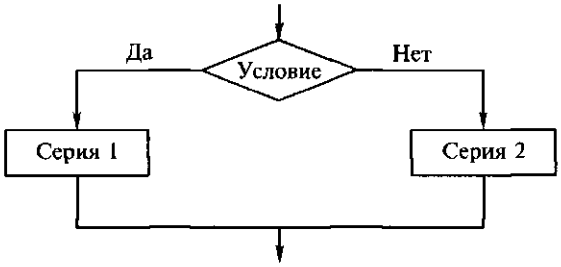


Рис. 1.5. Блок-схема команды ветвления

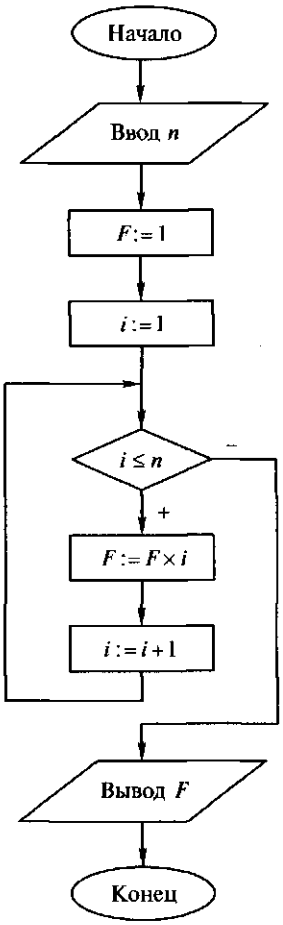


Рис. 1.6. Блок-схема алгоритма вычисления $n!$

Таблица 1.4. Трассировка алгоритма вычисления *n*!

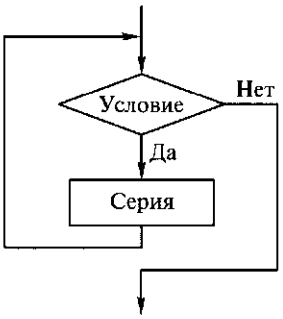
Шаг	*n*	*F*	*i*	Условие
1	3			
2		1		
3			1	
4				1 ≤ 3, да
5		1		
6			2	
7				2 ≤ 3, да
8		2		
9			3	
10				3 ≤ 3, да
11		6		
12			4	
13				4 ≤ 3, нет
14		Вывод		

```
алг факториал
цел n, i, F
нач ввод n
    F := 1; i := 1
    пока  i ≤ n, повторять
    нц    F := F × i
          i := i + 1
    кц
    вывод F
кон
```

Данный алгоритм имеет циклическую структуру. В нем использована структурная команда цикл «Пока», или цикла с предусловием. Блок-схема команды цикла «Пока» показана на рис. 1.7. На АЯ она имеет следующий вид:

```
пока      условие, повторять
нц
    серия
кц
```

Рис. 1.7. Блок-схема команды цикла «Пока»



Выполнение серии команд (тела цикла) повторяется пока условие цикла истинно. Когда условие становится ложным, выполнение цикла заканчивается. Служебные слова «нц» и «кц» обозначают соответственно начало и конец цикла.

Цикл с предусловием — это основная, но не единственная форма организации циклических алгоритмов: существует цикл с постусловием.

Вернемся к алгоритму решения квадратного уравнения, рассмотрев его со следующей позиции: если $a = 0$ — это уже не квадратное уравнение и его можно не решать. В данном случае будем считать, что пользователь ошибся при вводе данных, поэтому ввод следует повторить (рис. 1.8). Иначе говоря, в алгоритме будет предусмотрен контроль достоверности исходных данных с предоставлением пользователю возможности исправления ошибки. Наличие такого контроля — еще один признак хорошего качества программы.

На АЯ алгоритм решения квадратного уравнения с контролем ввода данных будет иметь следующий вид:

```
алг квадратное уравнение
вещ a, b, c, d, x1, x2
нач
    повторять
          ввод a, b, c
    до    a ≠ 0
          d := b^2 - 4ac
    если d ≥ 0
    то    x1 := (-b + √d)/(2a)
          x2 := (-b - √d)/(2a)
          вывод x1, x2
    иначе
          вывод «Нет вещественных корней»
    кв
кон
```

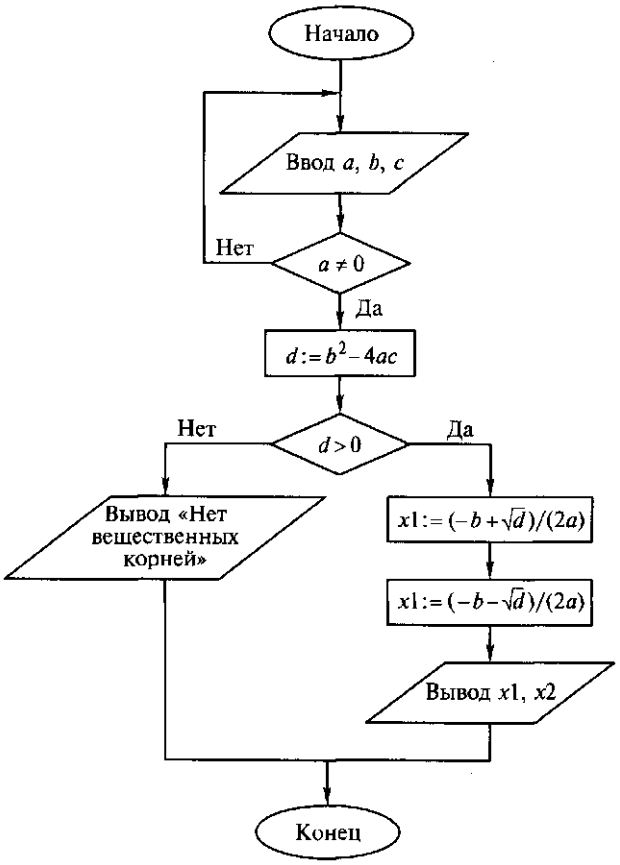


Рис. 1.8. Блок-схема алгоритма решения квадратного уравнения с контролем ввода данных

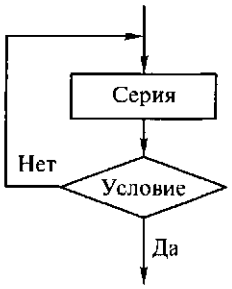
Блок-схема структурной команды цикла с постусловием или цикла «До» показана на рис. 1.9. На АЯ данную команду можно записать следующим образом:

```
повторять
    серия
до  условие
```

Здесь используется условие окончания цикла, т.е. когда оно становится истинным, цикл заканчивает работу.

Составим алгоритм решения следующей задачи: даны два натуральных числа M и N. Требуется вычислить их наибольший общий делитель — НОД (M, N).

Рис. 1.9. Блок-схема команды цикла «До»



Эта задача решается с помощью метода, известного под названием алгоритма Евклида. Идея этого метода основана на утверждении, что если $M > N$, то НОД (M, N) = НОД ($M - N$, N). Попробуйте доказать это самостоятельно. Другое утверждение, лежащее в основе решения данного алгоритма, очевидно: НОД (M, M) = M. Для «ручного» выполнения этот алгоритм можно описать в форме следующей инструкции:

1) если числа равны, взять их общее значение в качестве ответа. В противном случае продолжить выполнение алгоритма;

2) определить большее из чисел;

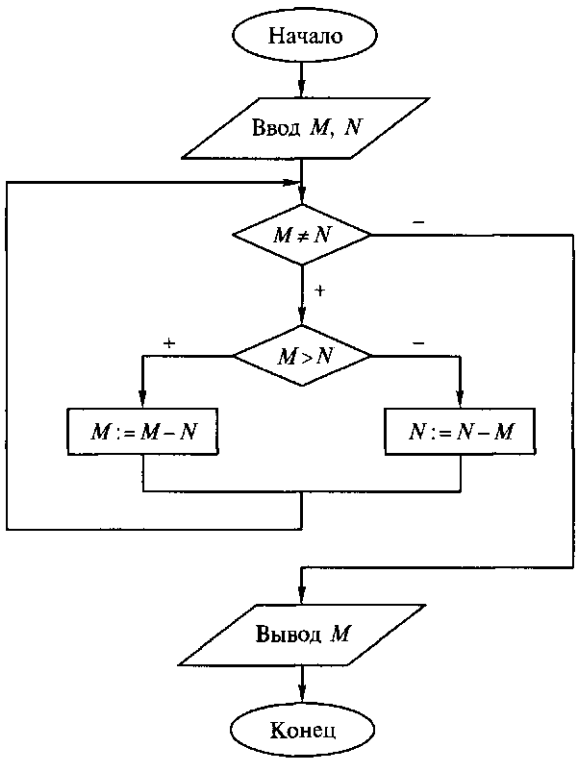


Рис. 1.10. Блок-схема алгоритма Евклида

3) заменить большее число разностью большего и меньшего значений;

4) вернуться к выполнению п. 1.

Блок-схема алгоритма Евклида приведена на рис. 1.10. На АЯ данный алгоритм можно записать следующим образом:

```
алг Евклид
цел M, N
нач ввод M, N
    пока  M ≠ N, повторять
    нц    если M > N
          то M := M-N
          иначе N := N-M
          кв
    кц
кон
```

Алгоритм Евклида имеет структуру цикла с вложенным ветвлением. Выполните самостоятельно трассировку этого алгоритма для случая $M = 18$, $N = 12$. В результате должен получиться НОД = 6.

1.4. ЛОГИЧЕСКИЕ ОСНОВЫ АЛГОРИТМИЗАЦИИ

Прямое отношение к программированию имеет дисциплина, называемая математической логикой, основу которой составляет алгебра логики, или исчисление высказываний. Под высказыванием понимается любое утверждение, в отношении которого можно однозначно сказать, истинно оно или ложно. Например, утверждение «Луна — спутник Земли» — истинно, «5 > 3» — истинно, «Москва — столица Китая» — ложно, «1 = 0» — ложно. ИСТИНА и ЛОЖЬ являются логическими значениями. Логические значения приведенных утверждений однозначно определены. Другими словами, их значения являются *логическими константами*.

Логическое значение неравенства $x < 0$, где x — переменная, является переменным, т. е. в зависимости от значения x оно может быть либо истинным, либо ложным. Таким образом вводится понятие *логической переменной*. В описаниях алгоритмов, а также в программах на различных языках программирования, логические переменные могут обозначаться символическими именами, которым соответствует логический тип данных. Иначе говоря, если известно, что A, B, X, Y и др. — переменные логического типа, это

означает, что они могут принимать только значение ИСТИНА или ЛОЖЬ.

Основы формального аппарата математической логики создал в середине XIX в. английский математик Джордж Буль. В его честь исчисление высказываний называется булевой алгеброй, а логические величины — булевскими.

Логическое выражение (*логическая формула*) — это простое или сложное высказывание. Сложное высказывание строится из простых с помощью *логических операций* (связок).

Имеются три основных логических операции: отрицание, конъюнкция (логическое умножение) и дизъюнкция (логическое сложение).

Отрицание обозначается в математической логике знаком «¬» и читается как НЕ. Это одноместная операция. Например, запись ¬ ($x = y$) читается следующим образом: НЕ (x равно y), т.е. значение будет истинным, если x не равно y, и ложным, если x равно y. Отрицание изменяет значение логической величины на противоположное.

Конъюнкция обозначается знаком & и читается как И. Это двухместная операция. Например, запись ($x > 0$) & ($x < 1$) означает, что данная логическая формула примет значение ИСТИНА, если $x \in (0, 1)$, и значение ЛОЖЬ — в противном случае. Следовательно, результатом конъюнкции является ИСТИНА, если истинны оба операнда.

Дизъюнкция обозначается знаком ∨, который читается как ИЛИ. Например, запись ($x = 0$) ∨ ($x = 1$) означает, что формула принимает истинное значение, если x — двоичная цифра (0 или 1). Следовательно, результатом дизъюнкции является ИСТИНА, если хотя бы один операнд имеет значение ИСТИНА.

Правила выполнения рассмотренных логических операций отражены в табл. 1.5, называемой *таблицей истинности* (здесь *А* и *В* — логические величины, а буквами «И» и «Л» обозначены соответственно значения ИСТИНА и ЛОЖЬ).

Таблица 1.5. Таблица истинности

№ п/п	*А*	*В*	НЕ *А*	*А* И *В*	*А* ИЛИ *В*
1	И	И	Л	И	И
2	И	Л	Л	Л	И
3	Л	И	И	Л	И
4	Л	Л	И	Л	Л

Последовательность выполнения операций в логических выражениях определяется старшинством операций: отрицание, конъюнкция, дизъюнкция. Кроме того, порядок выполнения операции в логических формулах можно менять с помощью скобок:

(*A* И *B*) ИЛИ (НЕ *A* И *B*) ИЛИ (НЕ *A* И НЕ *B*).

Рассмотрим для примера вычисление значения логической формулы

НЕ *X* И *Y* ИЛИ *X* И *Z*

при следующих значениях логических переменных: *X* = ЛОЖЬ, *Y* = ИСТИНА, *Z* = ИСТИНА.

Отметим цифрами порядок выполнения операций в заданном выражении:

① ② ④ ③

НЕ *X* И *Y* ИЛИ *X* И *Z*.

Используя таблицу истинности, выполним вычисление по шагам:

1) НЕ ЛОЖЬ = ИСТИНА;
2) ИСТИНА И ИСТИНА = ИСТИНА;
3) ЛОЖЬ И ИСТИНА = ЛОЖЬ;
4) ИСТИНА ИЛИ ЛОЖЬ = ИСТИНА.

Значение рассмотренной логической формулы — ИСТИНА.

Рассмотрим следующее утверждение: «Значение *X* находится в интервале от 0 до 1». Это утверждение можно записать в виде системы неравенств

$$0 \le X \le 1.$$

Соответствующее логическое выражение будет иметь вид

$$(X \ge 0) \text{ И } (X \le 1).$$

А теперь усложним задачу и запишем в виде логического выражения следующее утверждение: точка на плоскости с координатами (*X*, *Y*) находится в кольце с центром в начале координат, с внутренним радиусом, равным 1, и внешним радиусом, равным 2 (рис. 1.11). Это выражение будет иметь вид

$$(X^2 + Y^2 \ge 1) \text{ И } (X^2 + Y^2 \le 4).$$

Операция конъюнкции выполняется следующим образом: выбирается множество точек, удовлетворяющих первому неравен-

ству ($X^2 + Y^2 \geq 1$), и из полученного множества выделяется подмножество, удовлетворяющее второму неравенству ($X^2 + Y^2 \leq 4$), что и будет окончательным результатом. Можно также сказать, что искомое множество есть пересечение двух множеств, первое из которых удовлетворяет первому неравенству, а второе — второму. Конъюнкция — логическое умножение, т.е. операция, вычисляющая такое пересечение.

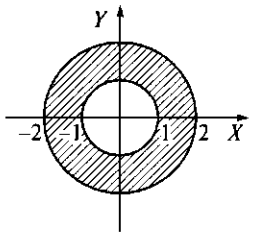


Рис. 1.11. Задание кольцевой области

Запишем в виде логического выражения следующее утверждение: точка на плоскости с координатами (X, Y) находится вне кольца с центром в начале координат с внутренним радиусом, равным 1, и внешним радиусом, равным 2 (см. рис. 1.11). Это выражение будет иметь вид

$$(X^2 + Y^2 < 1) \text{ ИЛИ } (X^2 + Y^2 > 4).$$

Выполнение операции дизъюнкции происходит следующим образом: выбирается множество точек, удовлетворяющих первому неравенству ($X^2 + Y^2 < 1$), и к нему прибавляется множество, удовлетворяющее второму неравенству ($X^2 + Y^2 > 4$). Так проясняется смысл другого названия операции дизъюнкции — логическое сложение, т.е. сложение двух множеств.

Последнюю задачу можно было решить иначе. Соответствующее логическое выражение можно записать как отрицание первого логического выражения, поскольку точки, находящиеся вне кольца, — это точки, которые НЕ принадлежат кольцу:

$$\text{НЕ } ((X^2 + Y^2 \geq 1) \text{ И } (X^2 + Y^2 \leq 4)).$$

Следовательно, справедливо равенство между двумя логическими выражениями:

$$\text{НЕ } ((X^2 + Y^2 \geq 1) \text{ И } (X^2 + Y^2 \leq 4)) = (X^2 + Y^2 < 1) \text{ ИЛИ}$$

$$(X^2 + Y^2 > 4).$$

Из этого примера становится понятным следующее правило преобразования: операция отрицания, примененная к двум неравенствам, связанным конъюнкцией, изменяет знаки этих неравенств на противоположные, а конъюнкцию — на дизъюнкцию (или дизъюнкцию на конъюнкцию).

Рассмотрим пример алгоритма, в котором используется сложное логическое выражение. По длинам трех сторон a, b, c треугольника вычислим его площадь.

Для решения данной задачи используется формула Герона:

$$\sqrt{p(p-a)(p-b)(p-c)},$$

где $p = (a + b + c)/2$ — полупериметр треугольника.

Исходные данные должны удовлетворять основному соотношению для сторон треугольника: длина каждой из сторон должна быть меньше суммы длин двух других его сторон.

Алгоритм вычисления площади треугольника имеет ветвящуюся структуру. Условие в структурной команде ветвления запишем в виде сложного логического выражения, которое позволит «отфильтровать» все варианты неверных исходных данных:

```
алг Герон
вещ A, B, C, P, S
нач ввод A, B, C
если (A > 0) И (B > 0) И (C > 0) И (A + B > C) И
     (B + C > A) И (A + C > B)
     то
          P := (A + B + C)/2
          S := корень (P *(P-A) * (P-B) * (P-C))
          вывод «Площадь =», S
     иначе вывод «Неверные исходные данные»
     кв
кон
```

1.5. ВСПОМОГАТЕЛЬНЫЕ АЛГОРИТМЫ И ПРОЦЕДУРЫ

В теории алгоритмов существует понятие *вспомогательного алгоритма*, т.е. алгоритма решения некоторой подзадачи из основной решаемой задачи. При этом алгоритм решения исходной задачи называется *основным*.

В качестве примера рассмотрим следующую задачу. Требуется составить алгоритм вычисления степенной функции с целым показателем: $y = x^k$, где k — целое число; $x \neq 0$.

В алгебре такая функция определена следующим образом:

$$x^n = \begin{cases} 1 & \text{при } n = 0; \\ \dfrac{1}{x^{-n}} & \text{при } n < 0; \\ x^n & \text{при } n > 0. \end{cases}$$

В данной задаче в качестве подзадачи можно рассматривать возведение числа в целую положительную степень.

Учитывая, что $1/x^{-n} = (1/x)^{-n}$, запишем основной алгоритм решения этой задачи:

```
алг степенная функция
цел n; вещ x, y
нач ввод x, n
    если n = 0
    то y: = 1
    иначе если n > 0
          то СТЕПЕНЬ (x, n, y)
          иначе СТЕПЕНЬ (1/x, -n, y)
          кв
    кв
    вывод y
кон
```

В основном алгоритме дважды повторяется команда обращения к вспомогательному алгоритму с именем СТЕПЕНЬ — алгоритму возведения вещественного основания в целую положительную степень посредством его многократного перемножения. Величины, стоящие в скобках в команде обращения к вспомогательному алгоритму, называются *фактическими параметрами*.

В учебном алгоритмическом языке вспомогательные алгоритмы оформляются в виде процедур. Запишем на АЯ процедуру СТЕПЕНЬ:

```
процедура СТЕПЕНЬ (вещ a, цел k, вещ z)
цел i
нач z := 1; i := 1
    пока i ≤ k, повторять
    нц    z := z × a
          i := i + 1
    кц
кон
```

Заголовок вспомогательного алгоритма начинается со слова «процедура», после которого следуют имя этой процедуры и в скобках — список формальных параметров. В этом списке перечисляются переменные аргументы и переменные результаты с указанием их типов. В данном примере a, k — формальные параметры-аргументы, z — параметр-результат. Следовательно, процедура СТЕПЕНЬ производит вычисления по формуле $z = a^k$.

В основном алгоритме вычисления степенной функции обращение к процедуре производится посредством указания ее имени и приведения далее в скобках списка фактических параметров.

Между формальными и фактическими параметрами процедуры должны выполняться правила соответствия:

- по количеству (сколько формальных, столько и фактических параметров);
- последовательности (первому формальному параметру соответствует первый фактический параметр, второму — второй и т.д.);
- типам (типы соответствующих формальных и фактических параметров должны совпадать).

Фактические параметры-аргументы могут быть выражениями соответствующего типа.

Обращение к процедуре инициирует следующие действия:

1) значения параметров-аргументов присваиваются соответствующим формальным параметрам;

2) выполняется тело процедуры (команды внутри процедуры);

3) значение результата передается соответствующему фактическому параметру, и происходит переход к выполнению следующей команды основного алгоритма.

В процедуре СТЕПЕНЬ нет команд ввода исходных данных и вывода результатов. Здесь присваивание начальных значений аргументам (a, n) производится через передачу параметров-аргументов, а присваивание результата переменной (y) происходит через передачу параметра-результата (z).

Таким образом, передача значений параметров процедур является третьим способом присваивания (наряду с командами присваивания и ввода).

Использование процедур позволяет строить сложные алгоритмы методом последовательной детализации.

1.6. ОСНОВЫ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

Прошло уже более полувека со времени появления первой ЭВМ. Все это время вычислительная техника стремительно развивалась. Изменялась элементная база ЭВМ, росли их быстродействие и объем памяти, изменялись средства интерфейса человека с машиной. Безусловно, эти изменения сказывались самым непосредственным образом на работе программиста. Определенный общепринятый способ производства чего-либо (в данном случае — программ) называют технологией, поэтому далее будем говорить о *технологии программирования*.

Для первых ЭВМ с «тесной» памятью и небольшим быстродействием основным показателем качества программы была ее экономичность по занимаемой памяти и времени счета. Чем программа получалась короче, тем класс программиста считался выше. Такое сокращение программы часто требовало больших усилий. Иногда программа получалась настолько «хитрой», что могла «перехитрить» самого автора: возвратившись через некоторое время к собственной программе и желая что-то изменить, программист мог запутаться в ней, забыв свою «гениальную идею».

Очень сложный алгоритм всегда увеличивает вероятность ошибки в программе, как и вероятность выхода из строя сложного технического устройства больше, чем простого.

Этапы изготовления программного продукта программистом, следующие:

1) проектирование;
2) кодирование;
3) отладка.

Проектирование заключается в разработке алгоритма будущей программы, например блок-схемы. *Кодирование* — это составление текста программы на языке программирования. *Отладка* осуществляется с помощью тестов, т.е. производится выполнение программы с некоторым заранее продуманным набором исходных данных, для которого известен результат. При этом чем сложнее программа, тем большее число тестов требуется для ее исчерпывающей проверки. Очень «хитрую» программу трудно протестировать исчерпывающим образом, так как всегда остается возможность не заметить какой-то «подводный камень».

С ростом памяти и быстродействия ЭВМ, с совершенствованием языков программирования и трансляторов с этих языков про-

блема экономичности программы становится менее острой. Все более важными качественными характеристиками программ становятся их простота, наглядность и надежность, а с появлением машин третьего поколения эти качества становятся основными.

В конце 60-х — начале 70-х гг. XX в. определяется дисциплина, получившая название структурного программирования. Ее появление и развитие связано с именами Э.В.Дейкстры, Х.Д.Милса, Д.Е.Кнута и др. Структурное программирование до настоящего времени остается основой технологии программирования. Соблюдение его принципов позволяет программисту быстро научиться создавать ясные, безошибочные, надежные программы.

В основе структурного программирования лежит теорема, которая была доказана в теории программирования: *алгоритм для решения любой логической задачи можно составить только из структур Следование, Ветвление, Цикл*, называемых базовыми алгоритмическими структурами.

Ранее уже приводились эти структуры. По сути, во всех рассмотренных примерах программ использовались принципы структурного программирования.

Следование — это линейная последовательность действий (рис. 1.12).

Каждая серия в такой последовательности может содержать в себе как простую команду, так и сложную структуру, но она обязательно имеет один вход и один выход.

Ветвление — это алгоритмическая альтернатива, которую можно записать в виде

```
если      условие
    то серия 1
    иначе серия 2
кв
```

Управление в этой структуре передается на одну из двух ветвей (серий команд) в зависимости от истинности или ложности условия. Затем происходит выход на общее продолжение (см. рис. 1.5).

Неполная форма ветвления имеет место при отсутствии ветви «иначе»:

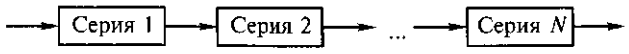


Рис. 1.12. Линейная структура программирования *Следование*

```
если      условие
   то серия
кв
```

Цикл — это повторение некоторой группы действий по условию. Различают два типа циклов. Первый тип циклической структуры — цикл с предусловием (см. рис. 1.7):

```
пока      условие, повторять
нц
   серия
кц
```

Здесь пока условие истинное, выполняется серия, образующая тело цикла.

Второй тип циклической структуры — цикл с постусловием (см. рис. 1.9):

```
повторять
   серия
до условие
```

Здесь тело цикла предшествует условию цикла и повторяет свое выполнение, если условие ложное. Повторение заканчивается, когда условие становится истинным.

Теоретически необходимым и достаточным для построения любого циклического алгоритма является цикл с предусловием, т.е. это более общий вариант цикла, чем цикл с постусловием. В самом деле, тело цикла «До» хотя бы один раз обязательно выполнится, так как проверка условия происходит после завершения его выполнения, а для цикла «Пока» возможен вариант, когда тело цикла не выполнится ни разу. Следовательно, в любом языке программирования можно было бы ограничиться использованием цикла «Пока», однако в ряде случаев применение цикла «До» оказывается более удобным, и поэтому он используется.

Иногда в литературе структурное программирование называют программированием без GOTO (без оператора безусловного перехода). Действительно, при таком подходе к программированию нет места безусловному переходу. Неоправданное использование в программах GOTO лишает их структурности, а значит, и всех связанных с этим положительных свойств алгоритма: прозрачности и надежности. Хотя во всех процедурных языках программирования этот оператор присутствует, для обеспечения структурно-

го подхода к программированию его употребления следует избегать.

Сложные алгоритмы состоят из соединенных между собой базовых структур. Соединение этих структур может выполняться двумя способами: *последовательным* и *вложенным*. Это аналогично ситуации в электротехнике, где любую сложную электрическую цепь можно разложить на последовательно и параллельно соединенные участки.

Однако вложенные алгоритмические структуры не являются аналогом параллельно соединенных электрических проводников. Здесь больше подходит аналогия с матрешками, помещенными друг в друга. Если блок, составляющий тело цикла, сам является циклической структурой, это значит, что имеют место вложенные циклы. В свою очередь внутренний цикл может иметь внутри себя еще один цикл и т. д. В связи с этим введено понятие *глубины вложенности циклов*. Точно также и ветвления могут быть вложенными друг в друга.

Структурный подход требует стандартного изображения блок-схем алгоритмов. Каждая базовая структура должна иметь один вход и один выход. Нестандартно изображенная блок-схема алгоритма плохо читается и теряет свою наглядность.

Примеры структурных блок-схем алгоритмов приведены на рис. 1.13. Такие блок-схемы легко читаются и хорошо воспринимаются зрительно, при этом каждой структуре можно дать название.

В учебном алгоритмическом языке, предназначенном для описания структурных алгоритмов, команда безусловного перехода отсутствует. Использование этого языка при обучении способствует «структурному воспитанию» программиста.

Наглядность структурам алгоритмов на АЯ придает структуризация вида их текста. Основной используемый для этого прием — сдвиги строк, которые должны подчиняться следующим правилам:

- конструкции одного уровня вложенности располагаются на одном вертикальном уровне (т. е. начинаются с одной позиции в строке);
- вложенные конструкции смещаются по строке на несколько позиций вправо относительно внешних для конструкций.

Структуры текстов алгоритмов на АЯ для показанных на рис. 1.13 блок-схем приведены в табл. 1.6.

Структурная методика алгоритмизации — это не только форма описания алгоритма, но и *способ мышления программиста*. Следу-

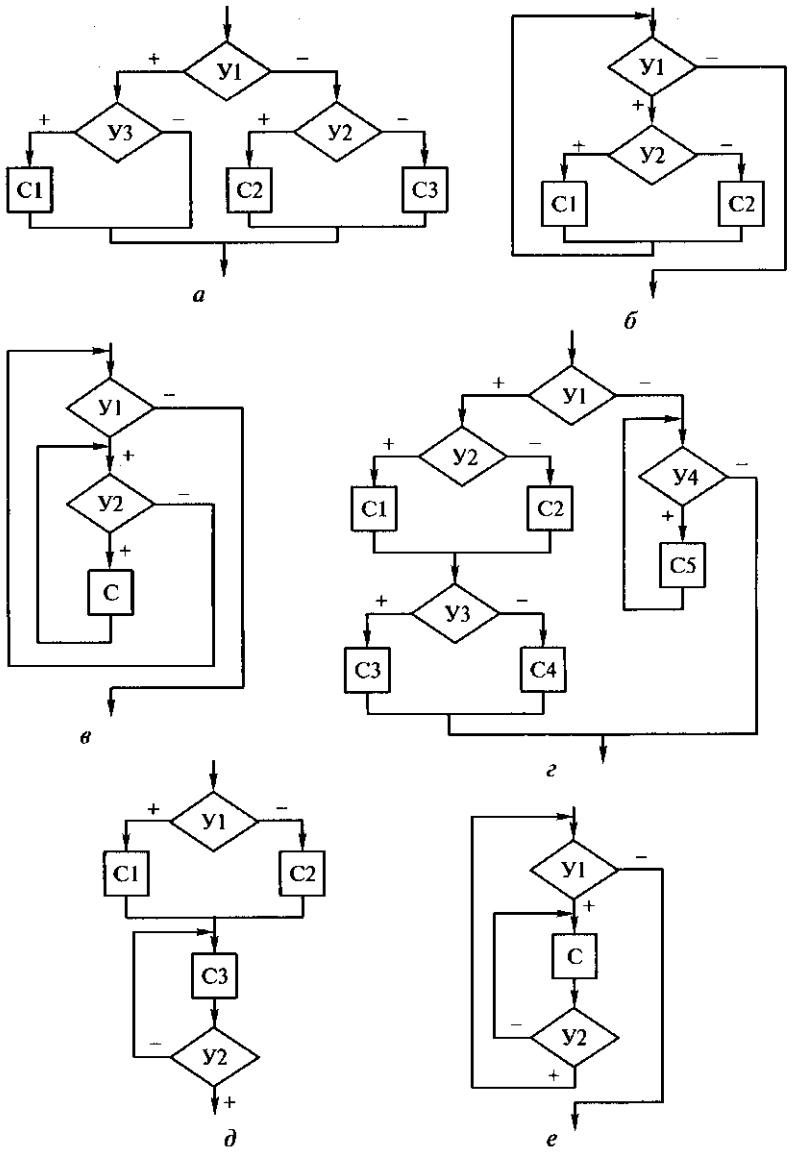


Рис. 1.13. Примеры структурных блок-схем алгоритмов:

а — вложенные ветвления с глубиной вложенности, равной единице; *б* — цикл с вложенным ветвлением; *в* — вложенные циклы «Пока» с глубиной вложенности, равной единице; *г* — ветвление с вложенной последовательностью ветвлений на положительной ветви и с вложенным циклом «Пока» на отрицательной ветви; *д* — последовательность ветвления и цикла «До»; *е* — вложенные внешний цикл «Пока» и внутренний цикл «До»; У — условие; С — серия

Таблица 1.6. Алгоритмы на АЯ, соответствующие схемам, представленным на рис. 1.13

Схема	Алгоритм	Схема	Алгоритм
а	**если** <У1> **то если** <У3> **то** <С1> **кв** **иначе если** <У2> **то** <С2> **иначе** <С3> **кв** **кв**	*б*	**пока** < У1>, **повторять** **нц** **если** <У2> **то** <С1> **иначе** <С2> **кв** **кц**
в	**пока** <У1>, **повторять** **нц** **пока** <У2>, **повторять** **нц** <С> **кц** **кц**	*г*	**если** <У1> **то если** <У> **то** <С1> **иначе** <С2> **кв** **если** <У3> **то** <С3> **иначе** <С4> **кв** **иначе пока** <У4>, **повторять** <С5> **кц** **кв**
д	**если** <У1> **то** <С1> **иначе** <С2> **кв** **повторять** С3 **до** <У2>	*е*	**пока** <У1>, **повторять** **не повторять** <С> **до** <У2> **кц**

ет стремиться составлять алгоритм из стандартных структур. Используя строительную аналогию, можно сказать, что структурная методика построения алгоритма подобна сборке здания из стандартных секций.

Еще одним важнейшим технологическим приемом структурного программирования является *декомпозиция решаемой задачи на подзадачи*, т.е. более простые для программирования части исходной задачи. Алгоритмы решения таких подзадач называются *вспомогательными*. При этом возможны два подхода к построению алгоритма:

- сверху вниз — сначала строится основной алгоритм, а затем — вспомогательные;
- снизу вверх — сначала составляются вспомогательные алгоритмы, а затем — основной.

Первый подход к построению алгоритма также называют методом последовательной детализации, а второй — сборочным методом.

При последовательной детализации сначала строится основной алгоритм и в него вносятся обращения к вспомогательным алгоритмам первого уровня. Затем составляются вспомогательные алгоритмы первого уровня, в которых могут присутствовать обращения к вспомогательным алгоритмам второго уровня, и т.д. Вспомогательные алгоритмы самого нижнего уровня состоят только из простых команд.

Метод последовательной детализации применяется при конструировании любых сложных объектов. Это естественная логическая последовательность мышления конструктора: постепенное углубление в детали. В программировании речь идет тоже о конструировании, но только не технических устройств, а алгоритмов. Достаточно сложный алгоритм другим способом построить практически невозможно.

Методика последовательной детализации позволяет организовать работу коллектива программистов над сложным проектом. Например, руководитель группы строит основной алгоритм, а разработку вспомогательных алгоритмов и написание соответствующих подпрограмм поручает своим сотрудникам. При этом участникам рабочей группы следует лишь договориться об интерфейсе (т.е. взаимосвязи) между своими программными модулями, а внутренняя организация программы — личное дело программиста.

Сборочный метод предполагает накопление и использование библиотек вспомогательных алгоритмов, реализованных в языках программирования в виде подпрограмм, процедур, функций.

1.7. РАЗВИТИЕ ЯЗЫКОВ И ТЕХНОЛОГИЙ ПРОГРАММИРОВАНИЯ

Язык программирования — это способ записи программ решения различных задач на ЭВМ в «понятной» для компьютера форме. Процессор компьютера непосредственно служит для восприятия языка машинных команд (ЯМК). Однако программы на ЯМК

разрабатывались лишь для ламповых ЭВМ первого поколения. Программирование на ЯМК — дело непростое. Программист должен знать числовые коды всех машинных команд и сам распределять память под команды программы и данные.

В 1950-х гг. появились первые средства автоматизации программирования — языки-автокоды. Позднее языки этого уровня стали называть ассемблерами. Языки типа автокода и ассемблера облегчили работу программистов. Переменные величины стали изображать символическими именами. Числовые коды операций заменили мнемоническими (словесными) обозначениями, которые легче запомнить. При этом язык программирования стал понятнее для человека, удалился от языка машинных команд. Чтобы компьютер мог исполнять программы на автокоде, требовался специальный переводчик — транслятор, т.е. системная программа, переводящая текст программы на автокоде в текст эквивалентной программы на ЯМК.

Компьютер, оснащенный транслятором с автокода, понимает автокод. В этом случае можно говорить о псевдоЭВМ (аппаратура + транслятор с автокода), языком которой является автокод. Языки типа автокода и ассемблера являются машинно-ориентированными, т.е. они настроены на структуру машинных команд конкретного компьютера. Разные компьютеры с различными типами процессоров имеют разный ассемблер. Языки программирования высокого уровня (ЯПВУ) являются машинно-независимыми, т.е. одна и та же программа на таком языке может выполнятся на ЭВМ разных типов, оснащенных соответствующим транслятором. Форма записи программ на ЯПВУ по сравнению с автокодом ближе к традиционной математической форме и естественному языку. Например, программа на языке Паскаль почти такая же, как на школьном алгоритмическом языке. Языки программирования высокого уровня легко поддаются изучению и хорошо поддерживают структурную методику программирования.

Первыми популярными языками высокого уровня, появившимися в 1950-х гг., были Фортран, Кобол (в США) и Алгол (в Европе). Это языки *первого поколения* ЭВМ. Фортран и Алгол были ориентированы на научно-технические расчеты математического характера. Кобол — это язык для программирования экономических задач, в котором слабее развиты математические средства, однако хорошо разработаны средства обработки текстов и лучше организован вывод данных в форме требуемого документа. Для первых ЯПВУ была характерна предметная ориентация.

Большое число языков программирования — *второго поколения* ЭВМ появилось в 1960—1970-х гг. (за всю историю существования ЭВМ их было создано более тысячи), однако распространились и выдержали испытания временем немногие.

В 1965 г. в Дартмутском университете был разработан язык Бейсик. По замыслу авторов это должен был быть простой язык, легко изучаемый, предназначенный для программирования несложных расчетных задач. Наибольшее распространение Бейсик получил на микроЭВМ и персональных компьютерах (ПК). На первых моделях школьных компьютеров программировать можно было только на языке Бейсик, однако это неструктурный язык, поэтому он плохо подходит для обучения качественному программированию. Справедливости ради следует заметить, что более поздние версии Бейсика для ПК, например QBasic, стали более структурными и по своим изобразительным возможностям приблизились к таким языкам, как Паскаль.

В эпоху ЭВМ *третьего поколения* большое распространение получил язык PL/1 (Program Language One), разработанный фирмой IBM. Это был первый язык, претендовавший на универсальность, т. е. на возможность решать любые задачи: вычислительные, обработки текстов, накопления и поиска информации. Однако PL/1 оказался слишком сложным языком. Для машин типа IBM 360/370 транслятор с языка PL/1 оказался недостаточно оптимальным и содержал ряд не выявленных ошибок. Для мини- и микроЭВМ язык PL/1 вообще не получил распространения. Однако линия на универсализацию языков программирования была поддержана: были разработаны универсальные версии старых языков — Алгол-68 и Фортран-77.

Значительным событием в истории языков программирования стало создание в 1971 г. Паскаля как учебного языка структурного программирования — языка *четвертого поколения* ЭВМ.

Широкое распространение языку Паскаль обеспечили ПК. Фирма Borland International, Inc (США) разработала для них систему программирования Турбо Паскаль. Турбо Паскаль — это не только язык и транслятор с него, но еще и *интегрированная среда программирования*, обеспечивающая пользователю удобство работы на языке Паскаль. Турбо Паскаль вышел за рамки учебного предназначения и стал языком профессионального программирования с универсальными возможностями. Транслятор с Турбо Паскаля по оптимальности создаваемых им программ близок к лидеру по этому качеству — транслятору с Фортрана. В силу своих до-

стоинств Паскаль стал источником многих основных современных языков программирования, например Ада, Модула-2 и др.

Язык программирования Си (английское название — C) создавался как инструментальный язык для разработки операционных систем, трансляторов, баз данных и других системных и прикладных программ. Так же, как и Паскаль, язык Си — это язык структурного программирования, но в отличие от Паскаля в нем заложены возможности непосредственного обращения к некоторым машинным командам и определенным участкам памяти компьютера.

Модула-2 — это еще один язык, предложенный Н. Виртом, являющийся развитием языка Паскаль и содержащий средства для создания больших программ.

ЭВМ будущего — *пятого* — поколения называют машинами искусственного интеллекта. Но прототипы языков для этих машин были созданы намного раньше их физического появления. Это языки ЛИСП и Пролог.

Первое упоминание о языке ЛИСП относится к 1958 г. Создан он на основе понятия рекурсивно определенных функций. А поскольку доказано, что любой алгоритм может быть описан с помощью некоторого набора рекурсивных функций, то ЛИСП по сути является универсальным языком. С его помощью на ЭВМ можно моделировать достаточно сложные процессы, в частности интеллектуальную деятельность людей. Парадигму программирования, реализованную в языке ЛИСП, называют *функциональным программированием*.

Язык Пролог разработан во Франции в 1972 г. также для решения проблем искусственного интеллекта. Этот язык позволяет в формальном виде описывать различные утверждения, логику рассуждений и заставляет ЭВМ давать ответы на заданные вопросы. В языке Пролог реализована *логическая парадигма программирования*.

В конце XX в. новым значительным направлением в развитии программного обеспечения ЭВМ стал объектно-ориентированный подход. Объекты — это структуры, объединяющие в единое целое данные и программы их обработки. Приобрели популярность объектно-ориентированные операционные системы (например, Windows), прикладные программы, а также *системы объектно-ориентированного программирования* (ООП).

Первым языком с элементами объектно-ориентированной технологии программирования был язык Симула-67. Развитие языка Си привело к появлению его объектно-ориентированной версии

под названием Си++. В Турбо Паскале, начиная с версии 5.5, также появились средства ООП.

В последнее время возникло новое направление в технологии программирования — *визуальное*. Стали создаваться системы визуального программирования, работающие под Windows, использование которых, в частности, позволяет легко и быстро программировать сложный графический интерфейс. Появились визуальные версии популярных языков программирования: Visual Basic, Delphi (развитие Турбо Паскаля), Borland C++ Builder и др.

Способы трансляции. Реализовать тот или иной язык программирования на ЭВМ — это значит создать транслятор с этого языка для данной ЭВМ. Существует два принципиально различных метода трансляции — *компиляция* и *интерпретация*. Для объяснения различия этих методов можно предложить следующую аналогию: лектор должен выступить перед аудиторией на незнакомом ей языке. При этом возможны:

- полный предварительный перевод, т.е. лектор заранее передает текст выступления переводчику, тот записывает перевод, размножает его и раздает слушателям (после чего лектор может и не выступать);
- синхронный перевод, т.е. лектор читает доклад, а переводчик одновременно с ним слово за словом переводит выступление.

Компиляция является аналогом полного предварительного перевода, а интерпретация — аналогом синхронного перевода. Транслятор, работающий по принципу компиляции, называется *компилятором*, а транслятор, работающий по принципу интерпретации, — *интерпретатором*.

При компиляции в память ЭВМ загружается программа-компилятор. Она воспринимает текст программы на ЯПВУ как исходную информацию, которая называется *исходным модулем*, загружаемым из текстового файла. После обработки компилятором программы на ЯПВУ получают *объектный модуль*. Следующий этап обработки программы, называемый редактированием связей (или линкованием), выполняет специальная программа — редактор связей. Этот редактор подключает к исходной программе необходимые для ее работы программные модули: процедуры, функции и др., в результате чего получают *загрузочный модуль* — программу на языке машинных команд, готовую к выполнению. При этом в оперативной памяти остается только программа на ЯМК, выполнение которой и дает искомые результаты.

Интерпретатор в течение всего времени работы программы находится во внутренней памяти — оперативном запоминающем устройстве (ОЗУ), куда помещается и программа на ЯПВУ. Интерпретатор в последовательности выполнения алгоритма считывает очередной оператор программы, переводит его в команды и тут же выполняет эти команды, после чего переходит к переводу и выполнению следующего оператора. При этом результаты предыдущих переводов в памяти не сохраняются, т.е. при повторном выполнении одной и той же команды она снова будет транслироваться.

При компиляции исполнение программы включает в себя три этапа: компиляцию, линкование и выполнение. При интерпретации, поскольку трансляция и выполнение совмещены, обработка программы на ЭВМ проходит в один этап. Однако откомпилированная программа выполняется быстрее, чем интерпретируемая, поэтому использование компиляторов удобнее для больших программ, требующих быстрого счета. Программы на языках Паскаль, Си, Фортран всегда компилируются. Бейсик чаще всего реализуется через интерпретатор. Некоторые языки имеют реализацию как через компилятор, так и через интерпретатор. В режиме интерпретации удобно отлаживать программу, а рабочие расчеты лучше осуществлять в режиме компиляции.

1.8. СТРУКТУРА И СПОСОБЫ ОПИСАНИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

Во всех языках программирования определены способы организации данных и действий над данными. Кроме того, существуют элементы языка, включающие в себя множество символов (алфавит), лексемы и другие изобразительные средства программирования. Несмотря на разнообразие языков программирования их изучение происходит приблизительно по одной схеме. Это обусловлено общностью структуры языков высокого уровня (рис. 1.14).

Описание в данном учебнике языков Паскаль и Delphi будет выполняться в соответствии с этой схемой.

Следует отметить сходство изучения естественных языков и языков программирования. Во-первых, для того чтобы читать и писать на иностранном языке, надо знать *алфавит* этого языка.

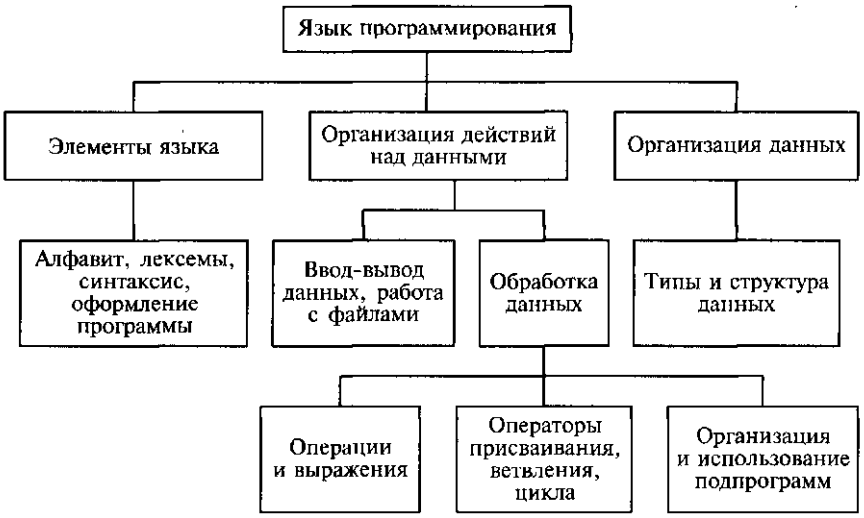


Рис. 1.14. Структура языка программирования высокого уровня

Во-вторых, следует знать правила написания слов и предложений, т.е. *синтаксис* языка. В-третьих, необходимо понимать смысл слов и фраз, чтобы адекватно реагировать на них. Например, в салоне самолета засветилось табло «Fasten belts!» (Застегните ремни!). Зная английскую грамматику, можно правильно прочитать эту фразу, однако не понять ее смысл и, следовательно, не выполнить соответствующих действий. Из грамотно написанных слов можно составить абсолютно бессмысленную фразу. Смысловое содержание языковой конструкции называется *семантикой*.

Любой язык программирования образуют три основные составляющие: алфавит, синтаксис и семантика.

Соблюдение правил в языке программирования должно быть более строгим, чем в разговорном языке. Человеческая речь содержит значительное количество избыточной информации, т.е. не расслышав какое-то слово, можно понять смысл фразы в целом. Слушающий или читающий человек может додумать, дополнить, исправить ошибки в воспринимаемом тексте.

Компьютер же — это автомат, воспринимающий все «всерьез». В текстах программ нет избыточности, и компьютер сам не исправит даже очевидной (с точки зрения человека) ошибки. Он может лишь указать место, которое «не понял» и вывести замечание о предполагаемом характере ошибки. Исправить же ошибку должен программист.

Для описания синтаксиса языка программирования также необходим какой-то язык, т.е. *метаязык* (надъязык), предназначенный для описания других языков. Наиболее распространенными метаязыками в литературе по программированию являются металингвистические *формулы Бэкуса—Наура* (язык БНФ) и *синтаксические диаграммы*. Далее мы будем использовать в основном язык синтаксических диаграмм, так как они более наглядны и легче воспринимаются. Однако, когда это будет удобно, будем применять и некоторые элементы языка БНФ.

В БНФ всякое синтаксическое понятие имеет вид формулы, состоящей из правой и левой частей, соединенных знаком «::=», смысл которого эквивалентен фразе «по определению есть». Левая часть формулы содержит имя определяемого понятия (метапеременную), заключенное в угловые скобки <>, а правая часть — формулу или диаграмму, определяющую все множество значений, которое может принимать метапеременная.

Синтаксис языка создается посредством последовательного усложнения понятий, т.е. сначала определяются простейшие (базовые) понятия, а затем все более сложные, включающие в себя предыдущие понятия как составляющие.

В такой последовательности, очевидно, конечным определяемым понятием должна быть «программа».

При записи метаформул приняты определенные соглашения. Например БНФ, определяющая понятие «двоичная цифра», имеет вид

```
<двоичная цифра> ::= 0 | 1
```

Знак «|» здесь эквивалентен слову «или» (либо).

Синтаксическая диаграмма двоичной цифры показана на рис. 1.15.

В этих диаграммах стрелки указывают на последовательность расположения элементов синтаксической конструкции, а кружками обведены символы, присутствующие в данной конструкции.

Понятие «двоичный код» как непустая последовательность двоичных цифр в БНФ описывается следующим образом:

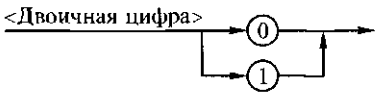


Рис. 1.15. Синтаксическая диаграмма двоичной цифры

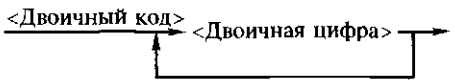


Рис. 1.16. Синтаксическая диаграмма двоичного кода

```
<двоичный код> ::= <двоичная цифра> | <двоичный
код><двоичная цифра>
```

Определение, в котором некоторое понятие определяется само через себя, называется *рекурсивным*. Рекурсивные определения характерны для БНФ.

Синтаксическая диаграмма двоичного кода представлена на рис. 1.16. Здесь возвратная стрелка обозначает возможность многократного повторения.

Очевидно, что синтаксическая диаграмма более наглядна, чем БНФ.

Синтаксические диаграммы были введены Н. Виртом и использованы для описания созданного им языка Паскаль.

УПРАЖНЕНИЯ

1. Определить значение логического выражения НЕ ($X > Z$) И НЕ ($X = Y$) при следующих значениях переменных:

а) $X = 3$, $Y = 5$, $Z = 2$;

б) $X = 0$, $Y = 1$, $Z = 19$;

в) $X = 5$, $Y = 0$, $Z = -8$;

г) $X = 9$, $Y = -9$, $Z = 9$.

2. Записать логические выражения (формулы), являющиеся истинными при следующих условиях:

а) точка с координатами (X, Y) принадлежит первой четверти единичного круга с центром в начале координат;

б) точка с координатами (X, Y) не принадлежит единичному кругу с центром в начале координат и принадлежит кругу с радиусом, равным 2, и с центром в начале координат (изобразите это графически).

3. Даны декартовы координаты трех вершин треугольника на плоскости. Составить алгоритм определения площади треугольника.

4. Дана скорость ракеты при выходе за пределы атмосферы Земли. Составить алгоритм определения движения ракеты после выключения двигателей. (Значения трех космических скоростей: 7,5 км/с; 11,2 км/с; 16,4 км/с.)

5. Даны три положительных числа. Составить алгоритм, определяющий, могут ли эти числа быть длинами сторон треугольника.

6. Пусть компьютер способен выполнять только две арифметические операции: сложение и вычитание. Составить следующие алгоритмы:

а) умножения двух целых чисел;

б) целочисленного деления двух чисел;

в) получения остатка от целочисленного деления двух чисел.

7. Построить алгоритм решения биквадратного уравнения, используя в качестве вспомогательного алгоритм решения квадратного уравнения.

8. Составить алгоритм нахождения НОД трех натуральных чисел, используя вспомогательный алгоритм нахождения НОД двух чисел.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каковы этапы решения задачи на компьютере с использованием программирования? Охарактеризуйте их. Проиллюстрируйте этапы постановки и формализации на примере следующей задачи: требуется вычислить время движения моторной лодки по реке между двумя пунктами.
2. Какие основные типы данных используются при программировании на компьютере?
3. Из каких команд составляется линейный алгоритм? Приведите пример задачи, решаемой с помощью линейного алгоритма.
4. Что такое алгоритмическая структура ветвления? В чем состоит разница между полным и неполным ветвлением? Приведите пример задачи, решаемой с помощью алгоритма с ветвящейся структурой.
5. Постройте ветвящийся алгоритм решения задачи, сформулированной в п. 1.
6. Что такое алгоритмическая структура цикла? В чем заключается разница между циклом с предусловием (цикл «Пока») и циклом с постусловием (цикл «До»)? Приведите пример задачи, решаемой с помощью циклического алгоритма. Составьте два варианта решения этой задачи: с циклом «Пока» и с циклом «До».
7. Что такое вспомогательный алгоритм (процедура)? Как описывается процедура на алгоритмическом языке? Как описывается обращение к процедуре в основном алгоритме? Проиллюстрируйте выделение подзадачи и использование процедуры для решения следующей задачи: найти площадь кольца по данным значениям внутреннего и внешнего радиусов.
8. Каковы правила вычисления логического выражения, содержащего логические операции? Приведите пример вычисления выражения с несколькими логическими операциями.
9. Что такое структурное программирование? Каковы основные принципы структурной методики построения алгоритмов?
10. Почему Ассемблеры называются машинно-ориентированными языками, а языки высокого уровня — машинно-независимыми языками программирования?
11. Что такое трансляция программы на языке высокого уровня? Чем различаются компиляция и интерпретация?

Глава 2

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ПАСКАЛЬ

Из данной главы вы узнаете:

- историю развития языка Паскаль;
- какую структуру имеет программа на Турбо Паскале;
- о концепции типов данных в Турбо Паскале;
- элементы языка — алфавит, операции, правила записи выражений;
- об организации ввода и вывода данных;
- правила выполнения оператора присваивания;
- способы программирования на Паскале алгоритмов с ветвящейся структурой;
- способы программирования циклов на Паскале;
- виды подпрограмм (функции и процедуры) и способы описания и использования подпрограмм;
- что такое рекурсивно-определенная подпрограмма (функция, процедура);
- о средствах программировании графики на Турбо Паскале;
- о структурных типах данных Турбо Паскаля (массивах, строках, множествах, записях, файлах) и способах их представления и обработки;
- что такое динамические величины и как создаются и обрабатываются динамические структуры данных — связанные списки;
- что такое модуль и как можно организовать библиотеку внешних подпрограмм.

Вы научитесь:

- программировать алгоритмы с линейной структурой и организацией ввода с клавиатуры и вывода на экран;
- программировать ветвящиеся алгоритмы с использованием операторов ветвления и выбора;
- использовать в программах все виды циклических операторов — цикл «Пока», цикл «До», цикл с параметром;
- программировать циклы с заданным числом повторений и итерационные циклы;
- программировать решение типовых задач с циклической структурой алгоритмов, в том числе табулирования функций, обработки конечных числовых последовательностей, обработки бесконечных сходящихся числовых последовательностей;
- программировать типовые задачи целочисленной арифметики;
- использовать в программах внутренние подпрограммы-функции и подпрограммы-процедуры;
- программировать решение типовых задач обработки одно- и двумерных массивов;
- программировать решение типовых задач обработки символьных строк;
- использовать в программах тип «множество»;
- программировать решение типовых задач обработки записей;
- программировать обмен данными с файлами, обработку данных в файлах;
- программировать несложные задачи обработки связанных списков;
- программировать получение на экране простых графических изображений — рисунков, чертежей, графиков функций;
- создавать небольшие программные модули (внешние библиотеки).

2.1. ПЕРВОЕ ЗНАКОМСТВО С ЯЗЫКОМ ПАСКАЛЬ

Структура программы на языке Паскаль. По определению стандартного языка Паскаль программа состоит из *заголовка* и

тела (*блока*), в конце которого следует *точка* — признак конца программы. В свою очередь, блок содержит *разделы описаний* и *раздел операторов*:

```
Program <имя программы>;
Label <раздел меток>;
Const <раздел констант>;
Type <раздел типов>;
Var <раздел переменных>;
Procedure (Function) <раздел подпрограмм>;
Begin
<раздел операторов>
End.
```


Раздел операторов имеется в любой программе и является основным. Разделы описаний могут не все содержаться в каждой программе.

В Турбо Паскале, в отличие от стандартного языка, возможно:

- отсутствие заголовка программы;
- следование разделов **Const**, **Type**, **Var**, **Label** друг за другом в любом порядке в разделе описаний и сколько угодно раз.

Примеры программ. Как уже говорилось, язык Паскаль разрабатывался Н.Виртом как учебный. Основной принцип, заложенный в нем, — это поддержка структурной методики программирования. На этом же принципе основывается псевдокод, который здесь будем называть алгоритмическим языком. По сути расхождение между АЯ и языком Паскаль заключается в следующем: АЯ — русскоязычный, Паскаль — англоязычный; синтаксис языка Паскаль определен строго и однозначно, а синтаксис АЯ — сравнительно свободно.

Запись программы на языке Паскаль похожа на английский перевод алгоритма, записанного на алгоритмическом языке. Сравните алгоритм деления простых дробей, записанный на АЯ, с соответствующей программой на Паскале:

```
алг деление дробей
цел a, b, c, d, m, n
нач ввод a,b,c,d
    m := a * d
    n := b * c
    вывод m,n
кон
```

```
Program Division;
Var a,b,c,d,m,n : Integer;
Begin ReadLn(a, b, c, d);
    m := a * d;
    n := b * c;
    WriteLn(m, n)
End.
```

Здесь учтено следующее математическое равенство:

$$\frac{a}{b} : \frac{c}{d} = \frac{a \cdot d}{b \cdot c}.$$

Эту программу можно понять, даже не заглядывая в учебник языка Паскаль, особенно, зная английский язык.

Заголовок программы начинается со слова **Program** (программа), за которым следует произвольное имя, придуманное программистом (division — деление). Раздел описания переменных начинается со слова **Var** (variables — переменные), за которым следует список переменных. Тип указывается после двоеточия словом **Integer** (целый). Начало и конец раздела операторов программы отмечаются словами **Begin** (начало) и **End** (конец). В конце программы *обязательно ставится точка.*

Ввод исходных данных с клавиатуры производится с помощью процедуры ReadLn (read line — считывать строку). На клавиатуре набирают четыре числа, отделяемые друг от друга пробелами, которые отражаются строкой на экране дисплея, и нажимают клавишу ввода.

Операторы присваивания в Паскале записываются так же, как в АЯ. Знак умножения обозначается звездочкой (*).

Вывод результатов на экран дисплея производится с помощью процедуры WriteLn (write line — писать в строку), которая выводит в строку два целых числа *m* и *n*. После этого курсор на экране переходит в начало следующей свободной строки и работа программы завершается.

Необходимо строго соблюдать правила правописания — синтаксис программы. В частности, в Паскале однозначно определено назначение знаков пунктуации:

- точка с запятой (;) ставится в конце заголовка программы, в конце раздела описания переменных, после каждого оператора. Перед словом **End** точку с запятой можно не ставить;
- запятая (,) является разделителем элементов во всевозможных списках (списке переменных в разделе описания, списке вводимых и выводимых величин).

Строгий синтаксис в языке программирования необходим, прежде всего, для транслятора. Транслятор — это программа, которая исполняется формально. Если, допустим, разделителем в списке переменных должна быть запятая, то любой другой знак в ней будет восприниматься как ошибка. Если точка с запятой является разделителем операторов, то транслятор в качестве оператора вос-

принимает всю часть текста программы от одной точки с запятой до другой. Следовательно, если не поставить точку с запятой между какими-то двумя операторами, транслятор будет принимать их за один, т.е. неизбежна ошибка.

Основное назначение синтаксических правил — это придание однозначного смысла языковым конструкциям. Если какая-то конструкция может трактоваться двусмысленно, значит, в ней есть ошибка. Поэтому лучше не полагаться на интуицию, а выучить правила языка.

Далее будет дано строгое описание синтаксических правил Паскаля, а пока для получения первоначального представления о языке обратимся еще к нескольким примерам программирования несложных алгоритмов.

«Оттранслируем» алгоритм вычисления факториала (*N*!) на языке Паскаль:

```
алг факториал
цел N, I, F
нач ввод(N)
  F := 1
  I := 1
  пока I <= N
  нц   F := F * I
       I := I + 1
  кц
  вывод F
кон
```

```
Program Factorial;
Var N, I, F: Integer;
Begin ReadLn(N);
  F := 1;
  I := 1;
  While I <= N Do
  Begin F := F * I;
        I := I + 1
  End;
  WriteLn(F)
End.
```

Из этого примера, во-первых, видно, как записывается на Паскале оператор цикла с предусловием (цикл «Пока»):

```
While <условие выполнения> Do <тело цикла>
```

While — пока, **Do** — делать. Если тело цикла содержит последовательность операторов, то говорят, что оно образует *составной оператор*, в начале и конце которого надо писать **Begin** и **End**. Служебные слова **Begin** и **End** часто называют *операторными скобками*, объединяющими несколько операторов в один составной. Если же тело цикла — один оператор (не составной), то операторных скобок не требуется. Тогда транслятор считает, что тело цикла заканчивается на ближайшем знаке «;».

Во-вторых, из примера видно, что в Паскале нет специальных слов для обозначения начала цикла (**нц**) и конца цикла (**кц**). На все случаи есть служебные слова **Begin** и **End**.

Рассмотрим еще один пример программы решения квадратного уравнения:

```
алг корни;
вещ a, b, c, d, x1, x2
нач

 повторять
  вывод «Введите a,b,c;
   a ≠ 0»)
  ввод a, b, c
  до a ≠ 0
  d := b^2 - 4ac

  если d ≥ 0
  то
  x1:=(-b + sqrt(d))/(2a)
  x2:=(-b - sqrt(d))/(2a)
  вывод x1, x2


  иначе вывод «Нет ве-
    щественных корней»
  кв
кон
```

```
Program Roots;
Var a,b,c,d,x1,x2 : Real;
Begin {Ввод данных с конт-
 ролем}
 Repeat
  WriteLn('Введите a, b, c;
   a<>0');
  ReadLn(a, b, c)
 Until a <> 0;
{Вычисление дискриминанта}
 d := b*b - 4*a*c;
 If d >= 0
 Then Begin {Есть корни}
 x1:=(-b + sqrt(d))/2/a);
 x2:=(-b - sqrt(d))/2/a);
 WriteLn('x1 =', x1,
   'x2 =', x2);
      End
  Else WriteLn('Нет ве-
    щественных корней')
End.
```

В этой программе по сравнению с приведенными ранее появилось много новых элементов. Имя вещественного типа данных в Паскале — Real.

Цикл с постусловием (цикл «До») программируется оператором

```
Repeat <тело цикла> Until <условие окончания>
```

Repeat — повторять, **Until** — до. Тело цикла может представлять собой как одиночный, так и составной оператор, однако употребления слов **Begin** и **End** не требуется, поскольку слова **Repeat** и **Until** сами выполняют роль операторных скобок.

Знак ≠ (не равно) в Паскале имеет обозначение <>, а знак ≥ (больше или равно) обозначается >=.

Правила записи арифметических выражений будут подробно рассмотрены немного позже. Используемая в формулах вычисления корней стандартная функция квадратного корня ($\sqrt{x}$), в Па-

скале записывается в виде sqrt(x). Порядок выполнения операций в выражении определяется скобками и старшинством операций. Старшинство операций определяется так же, как в алгебре. Операции одинаковые по старшинству выполняются в порядке их записи (слева направо).

Ветвление в Паскале программируется с помощью *условного оператора*, имеющего следующую форму:

```
If <условие> Then <оператор 1> Else <оператор 2>
```

If — если, **Then** — то, **Else** — иначе. Операторы 1 и 2 могут быть как одиночными, так и составными. Составной оператор следует заключать в операторные скобки **Begin** и **End**.

Так же, как и в алгоритмическом языке, возможно использование неполной формы условного оператора:

```
If <условие> Then <оператор>
```

Характерной чертой данной программы является использование в тексте комментариев. *Комментарий* — это любая последовательность символов, заключенных в фигурные скобки {...}. Можно также использовать в качестве ограничителей комментариев круглые скобки со звездочками (*...*). Комментарий не определяет никаких действий программы, а является лишь пояснительным текстом. Он может даваться в любом месте программы, где можно поставить пробел. Программист пишет комментарии не для компьютера, а для себя, придавая тексту программы большую ясность.

Хорошо откомментированные программы называют самодокументированными. В некоторых программах объем комментариев превышает объем вычислительных операторов.

Удачное использование комментариев — признак хорошего стиля программирования.

Для выполнения программы на ЭВМ ее следует *ввести в память*, *оттранслировать* и *исполнить*. Для этого на компьютере должны иметься специальные средства программного обеспечения, которые на ПК составляют систему Турбо Паскаль.

УПРАЖНЕНИЯ

«Оттранслировать» с алгоритмического языка на язык Паскаль:

а) алгоритм Евклида;

б) алгоритм выбора большего значения из трех;

в) алгоритм определения существования треугольника с заданными длинами сторон;
г) алгоритм умножения двух целых чисел, ограничиваясь только операциями сложения и вычитания;
д) алгоритм вычисления частного и остатка от целочисленного деления.

2.2. НЕКОТОРЫЕ СВЕДЕНИЯ О СИСТЕМАХ ПРОГРАММИРОВАНИЯ НА ПАСКАЛЕ

Система программирования — это совокупность системных средств, предназначенных для разработки программ на конкретном языке программирования. Современные системы программирования включают в себя:

- транслятор (компилятор или интерпретатор);
- интегрированную среду разработки программ;
- встроенный текстовый редактор;
- библиотеки стандартных программ и функций;
- средства отладки программ;
- встроенные справочные службы и пр.

В 1980-х гг. фирма Borland начала работу над модифицированной версией стандартного языка Паскаль, которая была названа Турбо Паскалем. Одновременно была разработана система программирования для языка Турбо Паскаль для персональных компьютеров серии IBM PC.

Впоследствии названием «Турбо Паскаль» стали обозначать также систему программирования. Система программирования Турбо Паскаль предназначалась для работы в среде операционной системы MS DOS.

Турбо Паскаль (как язык, так и система программирования) значительно изменялся за историю своего существования.

Первый вариант Турбо Паскаля фирма Borland выпустила в середине 1980-х гг. Затем было создано шесть модификаций системы, известных как версии 3.0, 4.0, 5.0, 5.5, 6.0, 7.0. Каждая из них представляет собой усовершенствование предыдущей версии. Все они создавались для семейства машин IBM PC и совершенствовались вместе с компьютерами.

Версия 3.0 была ориентирована на компьютеры малой мощности (IBM PC/XT). Разрабатываемые на ней программы имели ограничение на длину (не более 64 Кбайт); в ней не было средств раз-

дельной компиляции взаимосвязанных программ; операционная среда была весьма несовершенна. Большие изменения были внесены в версию 4.0. Появились современная диалоговая среда, средства раздельной компиляции программных модулей, мощная графическая библиотека.

Версия 5.0 отличалась в основном дальнейшими усовершенствованиями среды разработки, к которой был добавлен встроенный отладчик. В версию 5.5 были впервые включены средства поддержки ООП — современной технологии создания программ.

Главные отличия версии 6.0 — это новая среда, ориентированная на работу с устройством ввода типа «мышь» и использующая многооконный режим работы; объектно-ориентированная библиотека Turbo Vision, а также возможность включать в тексты программ команды Ассемблера.

Версия 7.0 не содержала каких-то принципиальных новшеств по сравнению с версией 6.0. Введены некоторые расширения языка программирования, а также дополнительные сервисные возможности системной оболочки. К старшим версиям Турбо Паскаля стали применять название Object Pascal.

Операционная система Windows поддерживает работу приложений, разработанных под MS DOS. Это относится и к Турбо Паскалю. Существует ряд современных систем программирования на Паскале, предназначенных для работы в среде операционной системы Windows. Среди них свободно распространяемые системы: Free Pascal, GNU Pascal. Язык и система программирования Pascal ABC, разработанные в Южном федеральном университете, широко используются в учебных заведениях России для обучения программированию.

В последующих подразделах настоящей главы будет описываться язык Паскаль в соответствии со старшими версиями Турбо Паскаля. При этом, в изучаемом объеме, обеспечивается преемственность основных средств языка со всеми современными реализациями Паскаля.

2.3. ЭЛЕМЕНТЫ ЯЗЫКА ТУРБО ПАСКАЛЬ

Алфавит. Алфавит языка Турбо Паскаль включает в себя буквы, цифры и специальные символы:

- латинские буквы от *A* до *Z* (прописные) и от *a* до *z* (строчные);

- цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
- шестнадцатеричные цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F;
- специальные символы + – * / = <> [ ]. , ( ) : ; { } ^ @ $ #.

Следующие комбинации специальных символов являются едиными символами, которые нельзя разделять пробелами:

:= знак присваивания;
>= больше или равно;
<= меньше или равно;
<> не равно;
(* *) ограничители комментариев (наряду с { });
(..) эквивалент [ ].

Пробелы — это символ пробела (ASCII-32) и все управляющие символы кода ASCII (от 0 до 31).

К специальным символам относятся также *служебные слова*, смысл которых определен однозначно и которые не могут быть использованы для других целей. Для языка — это единые символы.

Служебные слова языка Турбо Паскаль: absolute, and, array, begin, case, const, div, do, downto, else, end, external, file, for, forward, function, goto, if, implementation, in, inline, interface, interrupt, label, mod, nil, not, of, or, packed, procedure, program, record, repeat, set, shl, shr, string, then, to, type, unit, until, uses, var, while, with, xor.

Последние версии языка Турбо Паскаль содержат также ряд служебных слов, относящихся к работе с объектами и встроенным ассемблером.

Идентификаторы. Идентификатором называется символическое имя определенного программного объекта: константы, переменные, типы данных, процедуры, функции, программы. С помощью синтаксической диаграммы идентификатор можно представить в виде, показанном на рис. 2.1.

Расшифровать диаграмму можно следующим образом: идентификатор — это любая последовательность букв и цифр, начинаю-

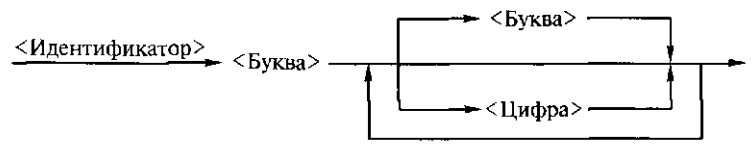


Рис. 2.1. Синтаксическая диаграмма идентификатора

щаяся с буквы. В Турбо Паскале к буквам приравнивается также знак подчеркивания.

Строчные и прописные буквы в идентификаторах и служебных словах не различаются. Например: max, MAX, MaX, mAx — одно и то же имя.

В Турбо Паскале длина идентификатора может быть произвольной, но значащими являются только первые 63 символа.

Комментарии. Конструкции следующего вида представляют собой комментарии и поэтому игнорируются компилятором:

```
{любой текст, не содержащий символ «}» }
(* любой текст, не содержащий символы «*)» *)
```

Буквы русского алфавита употребляются только в комментариях, литерных и текстовых константах.

Строка, начинающаяся с символов «{$» или «(*$», является директивой компилятора. За этими символами следует мнемоника команды компилятора.

2.4. КОНЦЕПЦИЯ ТИПОВ ДАННЫХ

Концепция типов данных является одной из центральных в любом языке программирования. С типом величины связаны три ее свойства: *форма внутреннего представления, множество принимаемых значений и множество допустимых операций.* Турбо Паскаль характеризуется большим разнообразием типов данных (рис. 2.2).

В стандартном Паскале отсутствует строковый тип данных. Кроме того, в Турбо Паскале целые и вещественные — это группы типов данных. В более поздних версиях Турбо Паскаля существуют процедурный тип данных и тип данных «объект».

Каждый тип данных имеет свой идентификатор.

Информация о простых типах данных, определенных в Турбо Паскале, представлена в табл. 2.1. Для вещественных типов данных в скобках указано количество сохраняемых значащих цифр мантиссы в десятичном представлении числа.

В стандартном Паскале из вещественных типов определен только тип Real, а из целых — Integer.

Типы данных Single, Double, Extended употребляются в Паскаль-программах только в том случае, если ПК снабжен сопроцессором «плавающей арифметики». (Для процессоров IBM PC, начиная с Intel-80486 и далее, это условие всегда выполняется.)

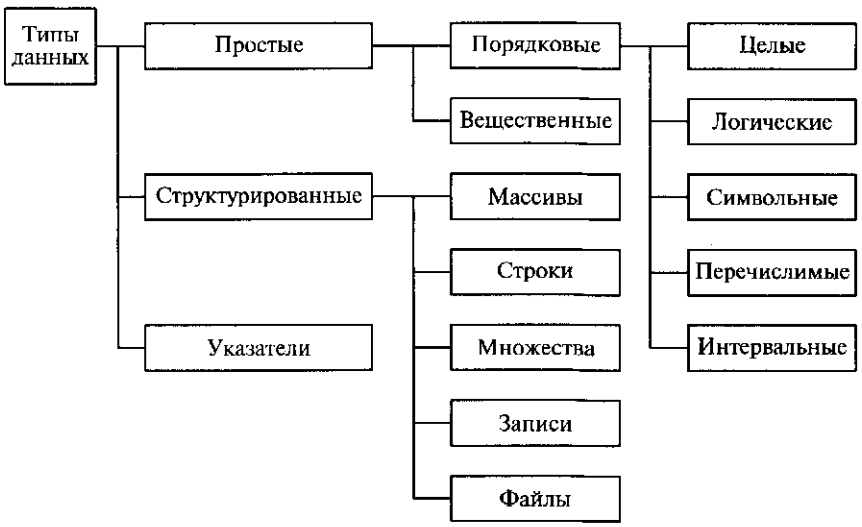


Рис. 2.2. Структура типов данных Турбо Паскаля

Тип данных называется *порядковым*, если состоит из счетного числа значений, которые можно пронумеровать. Отсюда следует, что для этого множества значений существуют понятия «следующий» и «предыдущий».

Описание переменных. Для всех переменных величин, используемых в программе, должны быть указаны их типы в *разделе переменных*. Структура раздела переменных показана на рис. 2.3.

Пример раздела переменных программы:

```
Var m, n, k : Integer;
    x, y, z : Real;
    Symbol : Char;
```

Константы. Тип константы определяется по контексту, т.е. по форме ее записи в программе.

Целые десятичные константы записывают в обычной форме целого числа со знаком или без знака. Например: 25, –24 712, 376.

Целые шестнадцатеричные константы записывают с префиксом «$». Они должны находиться в диапазоне от $00000000 до $FFFFFFFF.

Вещественные константы с фиксированной точкой записывают в обычной форме десятичного числа с дробной частью. Разделителем целой и дробной частей является точка. Например: 56.346, 0.000055, –345678.0.

Таблица 2.1. Типы данных Турбо Паскаля

Идентификатор	Длина, байт	Диапазон (множество) значений
Целые		
Integer	2	–32768..32767
Byte	1	0..255
Word	2	0..65535
Shortint	1	–128..127
Longint	4	–2147483648..2147483647
Вещественные		
Real	6	$2{,}9 \cdot 10^{-39}$...$1{,}7 \cdot 10^{38}$ (11...12)
Single	4	$1{,}5 \cdot 10^{-45}$...$3{,}4 \cdot 10^{38}$ (7...8)
Double	8	$5 \cdot 10^{-324}$...$1{,}7 \cdot 10^{308}$ (15...16)
Extended	10	$3{,}4 \cdot 10^{-4\,932}$...$1{,}1 \cdot 10^{4\,932}$ (19...20)
Логический		
Boolean	1	True, False
Символьный		
Char	1	Все символы кода ASCII

Вещественные константы с плавающей точкой имеют следующую форму:

```
<мантисса>E<порядок>
```

Здесь <мантисса> — целое или вещественное число с фиксированной точкой, <порядок> — целое число со знаком или без знака. Например: 7E–2 ($7 \cdot 10^{-2}$), 12.25E6 ($12{,}25 \cdot 10^{6}$), 1E–25 (10^{-25}).

Рис. 2.3. Структура раздела переменных

Символьная константа — это любой символ алфавита, заключенный в апострофы. Например: 'W', '!', '9'.

Логическая константа — это слова: True, False.

Строковая константа — это строка символов, заключенная в апострофы. Например: 'TurboPascal', 'Ответ: ', '35-45-79'. Максимальная длина строковой константы 255 символов.

Константе может быть поставлено в соответствие определенное имя, назначение которого производится в разделе констант программы. Например:

```
Const
  Max = 1000;
  G = 9.81;
  Cod = 'Ошибка';
```

Структура раздела констант показана на рис. 2.4. В Турбо Паскале допустимо также употребление *типизированных констант*. Типизированная константа аналогична переменной, которой задается начальное значение. Причем происходит это на этапе компиляции. Например:

```
Const  NumberCard  :  Integer  = 1267;
       Size        :  Real     = 12.67;
       Symbol      :  Char     = '*';
```

Описание типизированной константы приведено на рис. 2.5.

В Турбо Паскале имеется ряд имен, зарезервированных за определенными значениями констант, которые можно использовать без предварительного определения в программе (табл. 2.2).

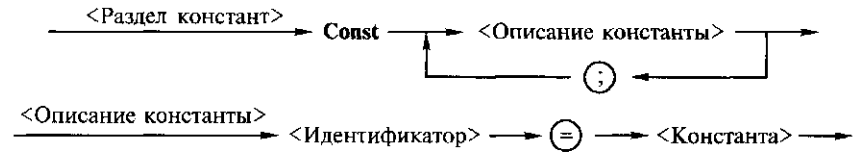


Рис. 2.4. Структура раздела констант

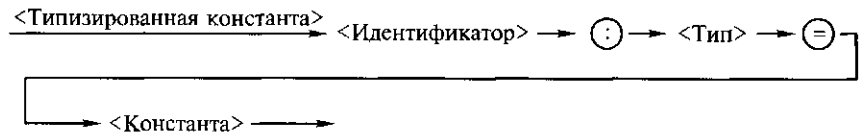


Рис. 2.5. Описание типизированной константы

Таблица 2.2. Зарезервированные константы Турбо Паскаля		
Идентификатор	Тип	Значение
True	Boolean	ИСТИНА
False	Boolean	ЛОЖЬ
MaxInt	Integer	32767

Типы данных пользователя. Один из принципиальных моментов языка Паскаль состоит в том, что пользователю разрешается определять свои типы данных. Причем типы данных пользователя всегда базируются на стандартных типах данных Паскаля.

Для описания типов данных пользователя в Паскале существует раздел типов, структура которого представлена на рис. 2.6.

Данные *перечисляемого типа* (рис. 2.7) задаются непосредственно перечислением всех значений, которые может принимать переменная данного типа.

Определенное имя типа данных затем используется для описания переменных. Например:

```
Type Gaz = (C, O, N, F);
     Metal = (Fe, Co, Na, Cu, Zn);
Var  G1, G2, G3: Gaz;
     Met1, Met2: Metal;
     Day: (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

Здесь Gaz и Metal — имена перечисляемых типов данных, которые ставятся в соответствие переменным G1, G2, G3 и Met1, Met2.

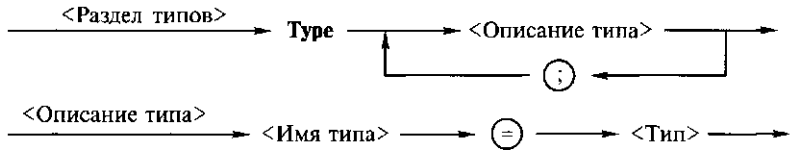


Рис. 2.6. Структура раздела типов данных

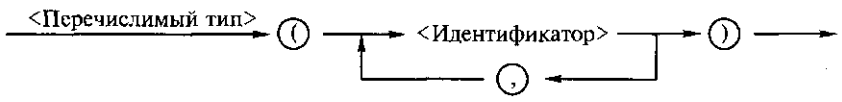


Рис. 2.7. Описание перечисляемого типа данных

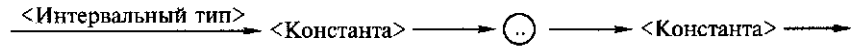


Рис. 2.8. Описание интервального типа данных

Переменной Day назначается перечисляемый тип данных, которому не присвоено имени.

Значения, входящие в перечисляемый тип данных, являются константами. Действия над ними подчиняются правилам, применяемым к константам. Каждое значение в перечисляемом типе занимает в памяти 2 байт, поэтому число элементов не должно превышать 65 535.

Перечисляемый тип данных — упорядоченное множество. Его элементы пронумерованы, начиная от 0 в порядке следования в описании.

В программе, в которой имеется приведенное ранее описание, возможно наличие следующего фрагмента:

```
If Day = Sun Then WriteLn('Ура! Сегодня выходной!');
```

Данные *интервального типа* (рис. 2.8) задаются как упорядоченное ограниченное подмножество некоторого порядкового типа.

Порядковый номер первой константы не должен быть больше номера второй константы в данных соответствующего базового типа.

При исполнении программы автоматически контролируется принадлежность значений переменной интервального типа установленному диапазону. При выходе из диапазона исполнение программы прерывается. Например:

```
Type  Numbers = 1..31;
      Alf = 'A'..'Z';
Var   Data: Numbers;
      Bukva: Alf;
```

2.5. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ, ФУНКЦИИ, ВЫРАЖЕНИЯ. ОПЕРАТОР ПРИСВАИВАНИЯ

К арифметическим типам данных относятся группы вещественных и целых типов данных. К ним применимы арифметические операции и операции отношений.

Различают операции над данными *унарные* (применимые к одному операнду) и *бинарные* (применимые к двум операндам).

Таблица 2.3. Бинарные операции Паскаля

Знак	Выражение	Тип операнда	Тип результата	Операция
+	*A* + *B*	R, R	R	Сложение
		I, I	I	
		I, R; R, I	R	
–	*A* – *B*	R, R	R	Вычитание
		I, I	I	
		I, R; R, I	R	
*	*A* * *B*	R, R	R	Умножение
		I, I	I	
		I, R; R, I	R	
/	*A*/*B*	R, R	R	Вещественное деление
		I, I	R	
		I, R; R, I	R	
div	*A* div *B*	I, I	I	Целое деление
mod	*A* mod *B*	I, I	I	Остаток от целого деления

Унарная арифметическая операция одна — это операция изменения знака следующего формата:

```
—<величина>.
```

Бинарные арифметические операции стандартного языка Паскаль описаны в табл. 2.3, где I — данные целого типа, а R — вещественного.

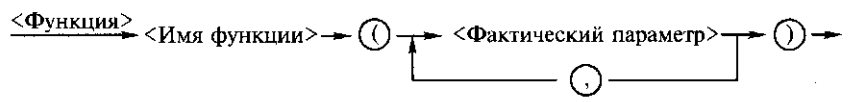


Рис. 2.9. Структура обращения к функции

Таблица 2.4. Стандартные математические функции Турбо Паскаля

Обращение	Тип аргумента	Тип результата	Функция
Pi	—	R	Число π = 3.1415926536E+00
abs (*x*)	I, R	I, R	Модуль аргумента *x*
arctan (*x*)	I, R	R	Арктангенс *x*, рад
cos (*x*)	I, R	R	Косинус *x*, рад
exp (*x*)	I, R	R	e^x — экспонента *x*
frac (*x*)	I, R	R	Дробная часть *x*
int (*x*)	I, R	R	Целая часть *x*
ln (*x*)	I, R	R	Натуральный логарифм *x*
random		R	Псевдослучайное число в интервале [0, 1]
random (*x*)	I	I	Псевдослучайное число в интервале [0, *x*]
round (*x*)	R	I	Округление *x* до ближайшего целого
sin (*x*)	I, R	R	Синус *x*, рад
sqr (*x*)	I, R	I, R	Квадрат *x*
sqrt (*x*)	I, R	R	Корень квадратный из *x*
trunc (*x*)	R	I	Ближайшее целое, не превышающее *x* по модулю

К арифметическим величинам могут быть применены стандартные функции языка Паскаль. Структура обращения к функции представлена на рис. 2.9.

Функция в выражении выступает как операнд. Например, в операторе присваивания

```
X := 2 * sin(A)/ln(3.5) + cos(C - D)
```

операндами являются три функции: sin, ln, cos, записываемые так же, как в математике. Аргументы называются фактическими параметрами, являются в общем случае выражениями арифметического типа и записываются в круглых скобках. Результатом вычисления функции является величина соответствующего типа.

Описания стандартных математических функций Турбо Паскаля содержит табл. 2.4.

Арифметическое выражение задает порядок выполнения действий над числовыми величинами. Арифметические выражения содержат арифметические операции, функции, операнды, круглые скобки. Одна константа или одна переменная — это простейшая форма арифметического выражения.

Например, математическое выражение

$$\frac{2a+\sqrt{0{,}5\sin(x+y)}}{0{,}2c-\ln(x-y)},$$

записанное по правилам языка Паскаль, будет иметь вид

```
(2 * A + sqrt(0.5 * sin(X + Y)))/(0.2 * C-ln(X-Y)).
```

Для верной записи арифметических выражений следует соблюдать определенные правила.

1. Все символы писать в строку, т. е. на одном уровне. Проставлять все знаки операций, не пропуская знак «*».

2. Не допускать записи двух знаков операций подряд, т. е. нельзя писать $A + -B$, следует писать $A + (-B)$.

3. Операции с более высоким приоритетом выполняют раньше операций с меньшим приоритетом. Порядок убывания приоритетов операций следующий:

- вычисление функции;
- унарная операция смены знака (–);
- *, /, div, mod;
- +, –.

4. Несколько записанных подряд операций с одинаковым приоритетом выполняют последовательно слева направо.

5. Часть выражения, заключенная в скобки, вычисляется в первую очередь. (Например, в выражении $(A + B) * (C - D)$ умножение производится после сложения и вычитания.)

6. Не следует записывать выражения, не имеющие математического смысла, например: деление на нуль, логарифм отрицательного числа и т. п.

Приведем пример. Арифметическое выражение, записанное по указанным правилам (цифрами в кружке указан порядок выполнения операций),

```
① ⑦   ④ ⑤    ③  ⑥   ②  ⑫    ⑪ ⑩       ⑧  ⑨
(1 + y) * (2 * x + sqrt(y)-(x + y))/(y + 1/(sqrt(x)- 4))
```

соответствует следующей математической формуле:

$$(1+y)\frac{2x+\sqrt{y}-(x+y)}{y+\dfrac{1}{x^2-4}}.$$

В Паскале нет операции или стандартной функции возведения числа в произвольную степень. Для вычисления x^y рекомендуется поступать следующим образом:

- если y — целое значение, следует использовать умножение, например: $x^3 \rightarrow x * x * x$. Для больших степеней следует использовать умножение в цикле;
- если y — вещественное значение, используется следующая математическая формула: $x^y = e^{y\ln(x)}$, запись которой на Паскале имеет вид

```
exp(Y * ln(x))
```

Очевидно, что при вещественном типе y недопустимо нулевое или отрицательное значение x. Для y целого типа такого ограничения нет.

Например, формула $\sqrt[3]{a+1} = (a+1)^{\frac{1}{3}}$, записанная на Паскале, будет иметь вид

```
exp(1/3 * ln(A + 1))
```

Выражение имеет целый тип, если в результате его вычисления получается величина целого типа. Выражение имеет вещественный тип, если результатом его вычисления является вещественная величина.

Арифметический оператор присваивания имеет структуру, представленную на рис. 2.10.

Например, это может быть запись вида

```
Y := (F * N - 4.5)/cos(X).
```

Порядок выполнения оператора присваивания рассматривался ранее. Следует только обратить внимание на следующее правило:

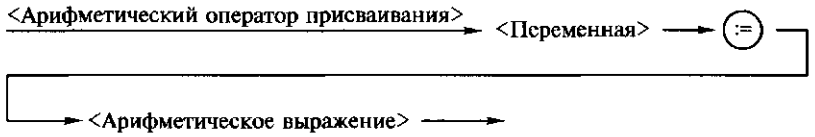


Рис. 2.10. Структура арифметического оператора присваивания

типы переменной и выражения должны быть одинаковыми. Исключением является случай, когда выражение имеет целый тип, а переменная — вещественный.

УПРАЖНЕНИЯ

1. Записать арифметические выражения на Паскале для следующих формул:

а) $a + bx + cyz$; б) $[(ax - b)x + c]x - d$; в) $\frac{a+b}{c} + \frac{c}{ab}$;

г) $\frac{x+y}{a_1}\frac{a_2}{x-y}$; д) $10^4\alpha - 3\frac{1}{5}\beta$; е) $\left(1+\frac{x}{2!}+\frac{y}{3!}\right) / \left(1+\frac{2}{3+xy}\right)$.

2. Записать математические формулы, соответствующие следующим выражениям на Паскале:

```
а) (p + q)/(r + s) - p * q/(r * s);
б) 1E3 + beta/(x - gamma * delta);
в) a/b * (c + d) - (a - b)/b/c + 1E - 8.
```

3. Почему в Паскале аргумент функции всегда записывают в скобках, например пишут ln(5), а не ln5?

4. Записать соответствующие арифметические выражения на Паскале для следующих формул:

а) $(1 + x)^2$; б) $\sqrt{1+x^2}$; в) $\cos^2 x^2$; г) $\log_2 \frac{x}{5}$;

д) arcsinx; е) $\frac{e^x + e^{-x}}{2}$; ж) $x^{\sqrt{2}}$; з) $\sqrt[3]{1+x}$;

и) $\sqrt{x^8 + 8^x}$; к) $\frac{xyz - 3{,}3\left|x + \sqrt[4]{y}\right|}{10^7 + \ln 4!}$; л) $\frac{\beta + \sin^2 \pi^4}{\cos 2 + \left|\mathrm{ctg}\,\gamma\right|}$.

5. Вычислить значения следующих выражений:

```
а) trunc(6.9);
б) trunc(6.2);
в) 20 div 6;
г) 2 div 5;
д) round(6.9);
е) round(6.2);
ж) 20 mod 6;
з) 2 mod 5;
и) 3 * 7 div 2 mod 7/3 - trunc(sin(1)).
```

6. Определить типы следующих выражений:

```
а) 1 + 0.0;
б) 20/4;
в) sqr(4);
г) sqrt(16);
д) sin(0);
е) trunc(-3.14).
```

7. Определить, какие из следующих операторов присваивания правильные, если y — вещественная переменная, а n — целая:

а) `y := n + 1;`
б) `n := y - 1;`
в) `n := 4.0;`
г) `y := trunc(y);`
д) `y := n div 2;`
е) `y := y div 2;`
ж) `n := n/2;`
з) `n := sqr(sqrt(n)).`

8. Поменять местами значения целых переменных *x* и *y*, не используя дополнительные переменные. Определить недостаток найденного алгоритма по сравнению с методом обмена значений через третью переменную. Можно ли применять данный алгоритм для вещественных чисел?

9. Присвоить целой переменной *h* значение цифры, стоящей в разряде сотен в записи положительного целого числа *k* (например, если *k* = 28 796, то *h* = 7).

10. Присвоить целой переменной *S* значение суммы цифр трехзначного целого числа *k*.

11. Определить, какую задачу решает следующая программа:

```
Program Test;
Type    Natur = 1..MaxInt;
Var     N : Natur;
        X : Real;
Begin   ReadLn(N);
        X := 0;
      While N > 0 Do
      Begin
        X := X + 1.0;
        N := N-1
      End;
      WriteLn(X)
End.
```

Указать, можно ли полученный результат получить более простым способом.

2.6. ВВОД ДАННЫХ С КЛАВИАТУРЫ И ВЫВОД НА ЭКРАН

Ввод данных — это передача информации от внешних устройств в оперативную память. Вводятся, как правило, исходные данные решаемой задачи.

Вывод данных — это передача данных из оперативной памяти на внешние носители (печать, дисплей, магнитные устройства и т.д.). Результаты решения любой задачи должны быть выведены.

Основными устройствами ввода-вывода персонального компьютера являются клавиатура и дисплей (экран монитора). Именно через эти устройства главным образом осуществляется диалог между человеком и ПК.

Процедура ввода с клавиатуры имеет следующий формат:

```
Read(<список ввода>)
```

Здесь <список ввода> — это последовательность имен переменных, разделенных запятыми, а Read (читать) — оператор обращения к стандартной процедуре ввода. Например:

```
Read(a, b, c, d)
```

При выполнении этого оператора работа компьютера прерывается, после чего пользователь набирает на клавиатуре значения переменных *a*, *b*, *c*, *d*, отделяя их друг от друга пробелами. При этом вводимые значения высвечиваются на экране. В конце набора нажимают клавишу <Enter>. Ввод значений должен выполняться в строгом соответствии с синтаксисом языка Паскаль. Например, при выполнении ввода в программе

```
Var T : Real;
    J : Integer;
    K : Char;
Begin
    Read(T, J, K);
```

на клавиатуре следует набрать

```
253.98 100 G [Enter].
```

Если в программе имеется несколько операторов Read, то данные для них вводятся потоком, т.е. после считывания значений переменных для одного оператора Read данные для следующего оператора до окончания строки считываются из той же строки на экране, что и для предыдущего затем происходит переход на следующую строку. Например, при выполнении ввода в программе

```
Var A, B : Integer;
    C, D : Real;
Begin
    Read(A, B);
    Read(C, D);
```

на клавиатуре следует набрать

```
18758 34[Enter] 2.62E-02 1.54E+01[Enter].
```

Оператор ввода с клавиатуры также может иметь вид

```
ReadLn(<список ввода>)
```

Здесь ReadLn (от read line) — считать строку. В отличие от оператора Read после считывания последнего в списке значений для одного оператора ReadLn данные для следующего оператора ReadLn будут считываться с начала новой строки. Если в предыдущем примере заменить оператор Read на ReadLn, т.е. записать

```
ReadLn(A, B);
ReadLn(C, D);
```

ввод значений будет происходить из двух строк:

```
18758 34 [Enter]
2.62E-02 1.54E+01 [Enter]
```

Оператор вывода на экран (обращение к стандартной процедуре вывода) имеет следующий формат:

```
Write(<список вывода>)
```

Здесь элементами списка вывода могут быть выражения различных типов (в частности, константы и переменные), например:

```
Write(234);                      {Выводится целая константа}
Write(A + B - 2);                {Выводится результат вычисления выражения}
Write(X, Summa, Arg1, Arg2);     {Выводятся значения переменных}
```

При выводе на экран нескольких чисел в строку они не отделяются друг от друга пробелами, об этом должен позаботиться программист. Пусть, например, $I = 1$; $J = 2$; $K = 3$. Тогда при выполнении оператора

```
Write(I, ' ', J, ' ', K);
```

на экране получим следующую строку: 1 2 3. Причем после вывода последнего символа курсор останется в той же строке и следующий ввод на экран будет начинаться с этой позиции курсора.

Процедура вывода на экран также может иметь вид

```
WriteLn(<список вывода>)
```

Здесь WriteLn (от Write line) — записать строку. Действие этого оператора отличается от Write тем, что после вывода последнего в списке значения происходит перевод курсора в начало следующей строки. Оператор WriteLn, записанный без параметров, выполняет перевод строки.

Форматы вывода. Список вывода может содержать указатели форматов вывода (форматы). Формат определяет представление выводимого значения на экране и отделяется от соответствующего ему элемента двоеточием. Если указатель формата отсутствует, машина выводит значение по определенному правилу, предусмотренному по умолчанию.

Далее кратко в справочной форме приведем правила и примеры бесформатного и форматированного вывода величин различных типов. Для представления списка вывода используем следующие обозначения:

I, P, Q — целочисленные выражения;
R — выражение вещественного типа;
B — выражение булевского типа;
Ch — символьная величина;
S — строковое выражение;
цифра;
* знак «+» или «–»;
_ пробел.

Форматы процедуры Write

I — выводится десятичное представление величины I, начиная с позиции расположения курсора:

Значение I	Оператор	Результат
134	Write(I)	134
287	Write(I,I,I)	287287287

I:P — выводится десятичное представление величины I в крайние правые позиции поля шириной P.

Значение I	Оператор	Результат
134	Write(I:6)	___134
312	Write((I + I):7)	_____624

R — в поле шириной 18 символов выводится десятичное представление величины R в формате с плавающей точкой (если R ≥ 0.0, используется формат ___#.##########E*##; если R < 0.0, формат имеет вид _-#.##########E*##):

Значение R	Оператор	Результат
715.432	Write(R)	__7.1543200000E+02
-1.919E+01	Write(R)	_-1.9190000000E+01

R:P — в крайние правые позиции поля шириной P символов выводится десятичное представление значения R в нормализованном формате с плавающей точкой. Минимальная длина поля вывода для положительных чисел составляет семь символов, для отрицательных — восемь. После точки выводится по крайней мере одна цифра:

Значение R	Оператор	Результат
511.04	Write(R:15)	5.110400000E+02
46.78	Write(-R:12)	-4.67800E+01

R:P:Q — в крайние правые позиции поля шириной P символов выводится десятичное представление значения R в формате с фиксированной точкой, причем после десятичной точки выводится Q цифр (0 ≤ Q ≤ 24), представляющих собой дробную часть числа (если Q = 0, то ни дробная часть, ни десятичная точка не выводятся; если Q > 24, то при выводе используется формат с плавающей точкой):

Значение R	Оператор	Результат
511.04	Write(R:8:4)	511.0400
-46.78	Write(R:7:2)	_-46.78

Ch:P — в крайнюю правую позицию поля шириной P выводится значение Ch:

Значение Ch	Оператор	Результат
'X'	Write(Ch:3)	_ _X
'!'	Write(Ch:2,Ch:4)	_!_ _ _!

S — начиная с позиции курсора выводится значение S:

Значение S	Оператор	Результат
'Day N'	Write(S)	Day N
'RRDD'	Write(S,S)	RRDDRRDD

S:P — значение S выводится в крайние правые позиции поля шириной P символов:

Значение S	Оператор	Результат
'Day N'	Write(S:10)	_ _ _ _ _Day N
'RRDD'	Write(S:5,S:5)	_RRDD_RRDD

B — выводится результат выражения B (True или False), начиная с текущей позиции курсора:

Значение B	Оператор	Результат
True	Write(B)	True
False	Write(B, Not B)	FalseTrue

B:P — в крайние правые позиции поля шириной P символов выводится результат булевского выражения:

Значение B	Оператор	Результат
True	Write(B:6)	_ _True
False	Write(B:6, Not B:7)	_False_ _ _True

2.7. УПРАВЛЕНИЕ СИМВОЛЬНЫМ ВЫВОДОМ НА ЭКРАН

Использование для вывода на экран только процедур Write и WriteLn обеспечивает программисту очень мало возможностей по управлению расположением на экране выводимого текста. Печать

текста может производиться только сверху вниз, слева направо. При этом невозможны возврат к предыдущим строкам, стирание напечатанного текста, изменение цвета символов и т.д.

Дополнительные возможности управления выводом на экран обеспечивают процедуры и функции модуля CRT. Для установления связи пользовательской программы с этим модулем перед разделами описаний должна быть поставлена строка

```
Uses CRT
```

Рассмотрим понятия, необходимые при работе с модулем CRT: режимы экрана, координаты позиции на экране, текстовое окно, цвет фона и цвет символа.

Режимы экрана. Во-первых, вывод на экран может происходить в текстовом или графическом виде (на графических дисплеях). Мы здесь будем говорить только о текстовом выводе.

Дисплеи могут быть монохроматическими (черно-белыми) и цветными. *Монохроматические дисплеи* могут работать только в черно-белом режиме, а *цветные* — как в черно-белом, так и в цветном. Кроме того, текстовые режимы различаются по числу символьных строк и столбцов, умещающихся на экране.

В модуле CRT каждый режим имеет определенный номер, за которым закреплено символическое имя (описанная константа). Для установки режима экрана используется процедура

```
TextMode(<номер режима>).
```

При обращении к процедуре <номер режима> может задаваться как числом, так и именем соответствующей константы. Например, эквивалентны следующие два оператора:

```
TextMode(1);
TextMode(CO40);
```

Как правило, исходный режим экрана, устанавливаемый по умолчанию, — CO80 (на цветных дисплеях).

Координаты позиции. Каждая символьная позиция на текстовом экране определена двумя координатами (*X*, *Y*). Координата *X* — позиция в строке. Крайняя левая позиция в строке имеет координату *X* = 1. Координата *Y* — номер строки, в которой находится символ. Строки нумеруются сверху вниз.

Например, в режиме 80×25 символ в верхнем левом углу экрана имеет координаты (1; 1), символ в нижнем правом углу экрана — (80; 25); символ в середине экрана — (40; 13).

Для установления курсора на экране в позицию с координатами (*X*, *Y*) в модуле CRT существует процедура

```
GoToXY(X,Y)
```

Здесь координаты курсора задаются выражениями типа Byte.

Приведем для примера программу, которая очищает экран и выставляет в центре экрана символ «*»:

```
Uses CRT;
Begin
    ClrScr;
    GoToXY(40,13);
    Write('*')
End.
```

Используемая здесь процедура ClrScr выполняет очистку экрана.

Текстовое окно. Прямоугольное пространство на экране, в которое производится вывод символов, называется *текстовым окном*. Положение этого окна определяется координатами верхнего левого и нижнего правого углов прямоугольника. Если окно занимает весь экран, то в режиме 80×25 его координаты (1; 1) и (80; 25). Это исходное окно, изменить положение и размер которого можно с помощью процедуры

```
Window(X1, Y1, X2, Y2)
```

Здесь аргументы — величины типа Byte; (*X*1, *Y*1) — координаты верхнего левого угла окна; (*X*2, *Y*2) — координаты правого нижнего угла окна.

После определения окна попытки вывода символов за его пределы окажутся безрезультатными. Повторное обращение к процедуре Window с новыми параметрами отменяет предыдущее назначение.

Управление цветом. На цветных дисплеях типа EGA, VGA и SVGA в текстовом режиме экрана можно использовать 16 цветов.

В модуле CRT объявлены константы, имена которых есть английские названия цветов, а соответствующие им значения — порядковые номера этих цветов.

Процедура назначения цвета фона

```
TextBackGround(Color)
```

Здесь аргумент — величина типа Byte, задающая номер цвета.
Процедура назначения цвета символа

```
TextColor(Color)
```

Если цвет фона назначается до очистки текстового окна, то после очистки оно «заливается» этим цветом. Если фон устанавливается после очистки экрана, то чистое окно будет иметь черный цвет (по умолчанию), а назначенный цвет фона будет устанавливаться в тех позициях, в которые выводятся символы.

Для примера приведем программу, которая по очереди откроет четыре окна, и каждое из этих окон «зальется» разным фоновым цветом:

```
Uses CRT;
Begin
    Window(1, 1, 40, 12);
    TextBackGround(White); ClrScr;
    Window(41, 1, 80, 12);
    TextBackGround(Red); ClrScr;
    Window(1, 13, 40, 25);
    TextBackGround(LightRed); ClrScr;
    Window(41, 13, 80, 25);
    TextBackGround(Green); ClrScr;
End.
```

По следующей программе на белом фоне в середине экрана будут выведены номера первых пятнадцати цветов, и каждый номер будет того цвета, который он обозначает:

```
Uses CRT;
Var I : Byte;
Begin
    TextBackGround(White);
    ClrScr;
    GoToXY(1, 12);
    For I := 0 To 14 Do
    Begin
          TextColor(I);
          Write(I:5)
    End
End.
```

Кратко опишем еще несколько процедур управления текстовым экраном из модуля CRT, не имеющих параметров:

- процедура ClrEOL — стирает часть строки от текущей позиции курсора до конца этой строки в окне. При этом положение курсора не изменяется;
- процедура DelLine — уничтожает всю строку с курсором. При этом нижние строки сдвигаются на одну вверх;
- процедура InsLine — вставляет пустую строку перед строкой, в которой стоит курсор;
- процедуры LowVideo, NormVideo, HighVideo — устанавливают соответственно режимы пониженной, нормальной и повышенной яркости символов.

Весьма полезной является функция KeyPressed из модуля CRT, при исполнении которой происходит опрос клавиатуры и определяется, не нажата ли какая-нибудь клавиша. В результате эта функция выдает логическое значение True, если нажата любая клавиша, и значение False — в противном случае. Часто данную функцию используют для организации задержки окна результатов на экране (после выполнения программы Турбо Паскаль вызывает на экран окно редактора), для чего перед концом программы записывают следующий оператор:

```
Repeat Until KeyPressed;
```

Это пустой цикл, который «крутится на месте» до нажатия какой-либо клавиши. В это время на экране выведено окно результатов. После нажатия клавиши значение KeyPressed станет равно True, цикл завершится, исполнение выйдет на конец программы и на экран вернется окно редактора. Этот прием можно использовать для задержки выполнения программы в любом ее месте.

В приведенную ранее программу получения на экране четырех разноцветных окон внесем следующее дополнение: после установки четырехцветного экрана выполнение программы останавливается и изображение сохраняется, а затем после нажатия любой клавиши экран возвращается в исходное состояние (80×25, черный фон, белые символы). Для этого перед концом программы следует добавить следующее:

```
Repeat Until KeyPressed;
Window(1, 1, 80, 25);
TextBackGround(Black);
ClrScr;
```

Описание других процедур и функций модуля CRT приведено в специальной литературе по Турбо Паскалю.

УПРАЖНЕНИЯ

1. Определить, что будет напечатано следующей программой, если в качестве исходных данных заданы числа 1.0 и –2.0:

```
Program Roots;
Var B, C, D : Real;
Begin
    Read(B, C);
    D := Sqrt(Sqr(B) - 4 * C);
    WriteLn('x1 =', (-B * D)/2,
            'x2 =', (-B - D)/2)
End.
```

2. Определить, что будет напечатано следующей программой при последовательном введении значений 3.4 и 7.9:

```
Program Less;
Var X : Real; T : Boolean;
Begin
    Read(X);
    T := X < Round(X);
    Read(X);
    T := T And (X < Trunc(X));
    WriteLn(T)
End.
```

3. Определить, что будет напечатано следующей программой при последовательном введении значений 36, –6 и 2 345:

```
Program ABC;
Var A, B : Integer;
Begin
    Read(A, B, A);
    WriteLn(A, B : 2, A : 5)
End.
```

4. Составить программу вычисления суммы двух целых чисел, которая будет вести диалог с пользователем в следующем виде (вместо отточий — вводимые и выводимые числа):

```
Введите два слагаемых
a =.....
b =.....
Результат вычислений:
a + b =.....
```

2.8. ЛОГИЧЕСКИЕ ВЕЛИЧИНЫ, ОПЕРАЦИИ, ВЫРАЖЕНИЯ

В подразд. 1.4 уже говорилось о логических величинах, логических операциях, логических выражениях. Напомним, что величина логического типа может принимать всего два значения: ИСТИНА и ЛОЖЬ.

В Паскале логические значения обозначаются служебными словами False (F) и True (T), а идентификатор логического типа — Boolean.

Кроме величин (констант и переменных) типа Boolean, логические значения False, True принимают результаты операций отношения.

Операции отношения осуществляют сравнение двух операндов и определяют, истинно или ложно соответствующее отношение между ними.

Структура операции отношения представлена на рис. 2.11, где

```
<знак отношения> ::= = (равно) | <> (не равно) |
> (больше) | < (меньше) | >= (больше или равно) |
<= (меньше или равно).
```

Приведем примеры записи отношений:

```
x < y; a + b >= c/d; abs(m-n) <= 1.
```

и примеры вычисления значений отношений:

Отношение	Результат
12 >= 12	True
56 > 10	True
11 <= 6	False

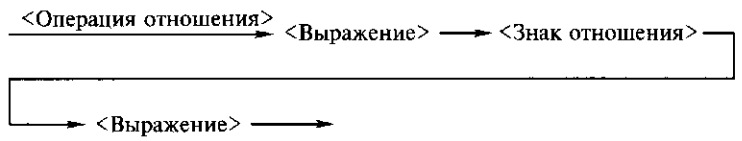


Рис. 2.11. Структура операции отношения

Логические операции выполняются над операндами булевского типа. Имеются три логические операции: **Not** — отрицание; **And** — логическое умножение (конъюнкция); **Or** — логическое сложение (дизъюнкция). Кроме этих трех обязательных операций, в Турбо Паскале имеется еще операция «Исключающее ИЛИ», обозначаемая служебным словом **Xor**. Это двухместная операция, которая в результате дает значение ИСТИНА, если оба операнда имеют разные логические значения.

Логические операции были перечислены в порядке убывания приоритетов. Результаты выполнения логических операций для различных значений операндов приведены в табл. 2.5.

Операции отношения имеют самый низкий приоритет, поэтому если операндами логической операции являются отношения, их следует заключать в круглые скобки. Например, математическому неравенству $1 \le x \le 50$ соответствует следующее логическое выражение:

```
(1 <= x) And (x <= 50)
```

Логическое выражение — это логическая формула, записанная на языке программирования. Логическое выражение состоит из логических операндов, связанных логическими операциями и круглыми скобками. Результатом вычисления логического выражения является булевская величина (False или True). Логическими операндами могут быть логические константы, переменные, функции, операции отношения. Один отдельный логический операнд является простейшей формой логического выражения.

Приведем примеры логических выражений, в которых *d*, *b*, *c* — логические числа; *x*, *y* — вещественные; *k* — целая переменная:

```
1) x < 2 * y;    2) True;          3) d;
4) Odd(k);       5) Not Not d;     6) Not (x > y/2);
7) d And (x <> y) And b;
8) (c Or d) And (x = y) Or Not b.
```

Таблица 2.5. Результаты выполнения логических операций

A	*B*	Not *A*	*A* And *B*	*A* Or *B*	*A* Xor *B*
T	T	F	T	T	F
T	F	F	F	T	T
F	F	T	F	F	F
F	T	T	F	T	T

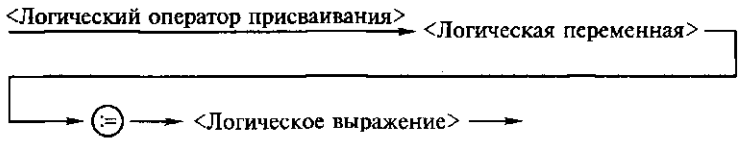


Рис. 2.12. Структура логического оператора присваивания

При d = True, b = False, c = True, x = 3.0, y = 0.5, k = 5 результаты вычислений будут следующими:

1) False; 2) True; 3) True; 4) True;
5) True; 6) False; 7) False; 8) True.

В данном примере использована логическая функция Odd(x). Это функция от целого аргумента x, которая принимает значение True, если значение x нечетное, и значение False, если оно четное.

Логический оператор присваивания имеет структуру, представленную на рис. 2.12.

Приведем примеры логических операторов присваивания:

1) `d := True;`
2) `b := (x > y) And (k <> 0);`
3) `c := d Or b And Not(Odd(k) And d).`

2.9. ФУНКЦИИ, СВЯЗЫВАЮЩИЕ РАЗЛИЧНЫЕ ТИПЫ ДАННЫХ

Список стандартных функций, обеспечивающих связь между различными типами данных, приведен в табл. 2.6.

Функции Ord, Pred и Succ применимы только к порядковым типам данных, т.е. из простых типов данных ко всем, кроме вещественных.

Функция Ord, применяемая к целому числу, дает его собственное значение. Например:

```
Ord(-35) = -35; Ord(128) = 128
```

Если аргумент целый, то, например, оператор y := Pred(x) эквивалентен $y := x - 1$, а оператор y := Succ(x) эквивалентен $y := x + 1$.

Для аргумента символьного типа эти функции дают соответственно предыдущий и следующий символы в таблице внутренней

Таблица 2.6. Стандартные функции, связывающие различные типы данных

Обращение	Тип аргумента	Тип результата	Действие
Ord (*x*)	Любой порядковый	I	Дает порядковый номер значения *x* в его типе
Pred (*x*)	То же	Тот же, что у *x*	Дает предыдущее по отношению к *x* значение в его типе
Succ (*x*)	»	Тот же, что у *x*	Дает следующее по отношению к *x* значение в его типе
Chr (*x*)	Byte	Char	Дает символ с порядковым номером *x*
Odd (*x*)	I	Boolean	Дает True, если *x* — нечетное число, и False, если *x* — четное

кодировки. Поскольку латинский алфавит всегда упорядочен по кодам, т.е.

```
Ord('a') < Ord('b') < ... < Ord('z'),
```

то, например, можно записать

```
Pred('b') = 'a'; Succ('b') = 'c'.
```

То же относится и к цифровым литерам:

```
Pred('5') = '4'; Succ('5') = '6'.
```

Функция Chr(*x*) является обратной к функции Ord(*x*), если *x* — символьная величина. Например, для кода ASCII справедлива запись:

```
Ord('a') = 97; Chr(97) = 'a'.
```

Эту их «взаимообратность», если *x* — символьная величина, можно выразить формулой

```
Chr(Ord(x)) = x.
```

В некоторых случаях возникает задача преобразования символьного представления числа в числовое. Например, получить из литеры '5' целое число 5 можно следующим образом:

```
N := Ord('5') - Ord('0').
```

Здесь *N* — целая переменная и использован тот факт, что код литеры '5' на пять единиц больше кода '0'.

Булевский тип данных также является порядковым. Порядок расположения двух его значений следующий: False, True. Тогда справедливы следующие отношения:

```
Ord(False) = 0; Succ(False) = True;
Ord(True) = 1;  Pred(True) = False.
```

Приведем интересный пример. Пусть *x*, *y*, *z* — вещественные переменные. Требуется определить, какую задачу решает следующий оператор:

```
z := x * Ord(x >= y) + y * Ord(y > x).
```

Ответ: $z = \max(x, y)$, т.е. данную задачу можно решить без использования условного оператора if.. then.. else.

УПРАЖНЕНИЯ

1. Вычислить значения следующих логических выражений:
   а) `K mod 7 = K div 5 - 1` при $K = 15$;
   б) `Odd(Trunc(10 * P))` при $P = 0.182$;
   в) `Not Odd(n)` при $n = 0$;
   г) `t And (P mod 3 = 0)` при t = True, $P = 10101$;
   д) `(x * y <> 0) And (y > x)` при $x = 2$, $y = 1$;
   е) `a Or Not b` при a = False, b = True.
2. Определить какое значение получит логическая переменная *d* при a = True и $x = 1$ после выполнения следующих операторов присваивания:
   а) `d := x < 2;`
   б) `d := Not a Or Odd(x);`
   в) `d := Ord(a) <> x.`
3. Написать оператор присваивания, в результате выполнения которого логическая переменная *t* получит значение True, если следующее утверждение истинно, и значение False — в противном случае:
   а) из чисел *x*, *y*, *z* только два равны между собой;
   б) *x* — положительное число;
   в) каждое из чисел *x*, *y*, *z* положительное;
   г) только одно из чисел *x*, *y*, *z* положительное;
   д) p делится без остатка на *q*;
   е) цифра 5 входит в десятичную запись трехзначного целого числа *k*.

2.10. ПРОГРАММИРОВАНИЕ ВЕТВЯЩИХСЯ АЛГОРИТМОВ

Алгоритмическая структура ветвления программируется в Паскале с помощью условного оператора, имеющего вид

```
If <Условие> Then <Оператор 1> Else <Оператор 2>;
```

Кроме того, возможно использование неполной формы условного оператора:

```
If <Условие> Then <Оператор>;
```

Строгое описание условного оператора в форме синтаксической диаграммы показано на рис. 2.13.

Условием в условном операторе является логическое выражение, которое вычисляется в первую очередь. Если его значение равно True, то будет выполняться <Оператор 1> (после Then), если же его значение равно False, будет выполняться <Оператор 2> (после Else) для полной формы или сразу оператор, следующий после условного, для неполной формы (без Else).

Пример 2.1. Требуется составить программу вычисления площади треугольника по длинам трех сторон *a*, *b*, *c*.

Для решения задачи используется формула Герона

$$\sqrt{p(p-a)(p-b)(p-c)},$$

где $p = (a + b + c) / 2$ — полупериметр треугольника.

Исходные данные должны удовлетворять основному соотношению для сторон треугольника: длина каждой стороны должна быть меньше суммы длин двух других сторон.

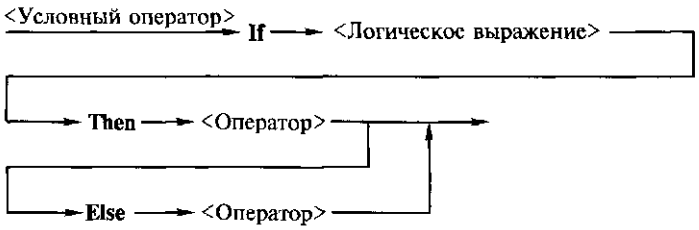


Рис. 2.13. Синтаксическая диаграмма условного оператора

Имея возможность в одном условном операторе записывать достаточно сложные логические выражения, можно сразу «отфильтровать» все варианты неверных исходных данных:

```
Program Geron;
Var A, B, C, P, S : Real;
Begin
  WriteLn('Введите длины сторон треугольника:');
  Write('a = '); ReadLn(A);
  Write('b = '); ReadLn(B);
  Write('c = '); ReadLn(C);
  If (A > 0) And (B > 0) And (C > 0) And (A + B > C)
       And (B + C > A) And (A + C > B)
  Then  Begin
          P := (A + B + C)/2;
          S := sqrt(P * (P-A) * (P-B) * (P-C));
          WriteLn('Площадь = ', S)
        End
  Else WriteLn('Неверные исходные данные')
End.
```

Пример 2.2. Требуется составить программу решения квадратного уравнения.

Алгоритм решения данной задачи подробно рассматривался в подразд. 1.3 (см. рис. 1.8). Алгоритм имеет структуру вложенных ветвлений. Его описание на алгоритмическом языке также приведено в подразд. 1.3. Программа на Паскале будет иметь вид

```
Program Roots;
Var A, B, C, D, X1, X2 : Real;
Begin
  WriteLn('Введите коэффициенты
         квадратного уравнения:');
  Write('a ='); ReadLn(A);
  Write('b ='); ReadLn(B);
  Write('c ='); ReadLn(C);
  If (A = 0)
  Then    If (B = 0)
          Then  If (C = 0)
                Then WriteLn('Любое X - решение')
                Else WriteLn('Нет решений')
          Else
          Begin
                X := -c/b;
                WriteLn('X = ', X)
          End
  Else
```

```
  Begin d := b * b - 4 * a * c;
        If (d < 0)
        Then WriteLn('Нет вещественных корней')
        Else
        Begin
             X1 := (-b + sqrt(d))/2/a;
             X2 := (-b - sqrt(d))/2/a;
             WriteLn('X1 = ', X1);
             WriteLn('X2 = ', X2)
        End
  End
End.
```

Пример 2.3. Требуется составить программу перевода пятибалльной оценки в ее наименование: 5 — отлично, 4 — хорошо, 3 — удовлетворительно, 2 — неудовлетворительно.

Блок-схема алгоритма решения задачи приведена на рис. 2.14.

Этот алгоритм имеет структуру вложенных ветвлений и соответствующую программу на Паскале можно записать следующим образом:

```
Program Marks_2;
Var N : Integer;
Begin
  WriteLn('Введите оценку:'); ReadLn(N);
  If (N = 5)
  Then WriteLn('Отлично')
  Else If (N = 4)
       Then WriteLn('Хорошо')
       Else If (N = 3)
            Then WriteLn('Удовлетворительно')
            Else If (N = 2)
                 Then WriteLn('Неудовлетворительно');
                 Else WriteLn('Неверная оценка')
End.
```

Структуру вложенных ветвлений можно запрограммировать с помощью одного оператора выбора, имеющегося в языке Паскаль:

```
Program Marks;
Var N: Integer;
Begin
    WriteLn('Введите оценку:');
    ReadLn(N);
    Case N Of
          5: WriteLn('Отлично');
          4: WriteLn('Хорошо');
```

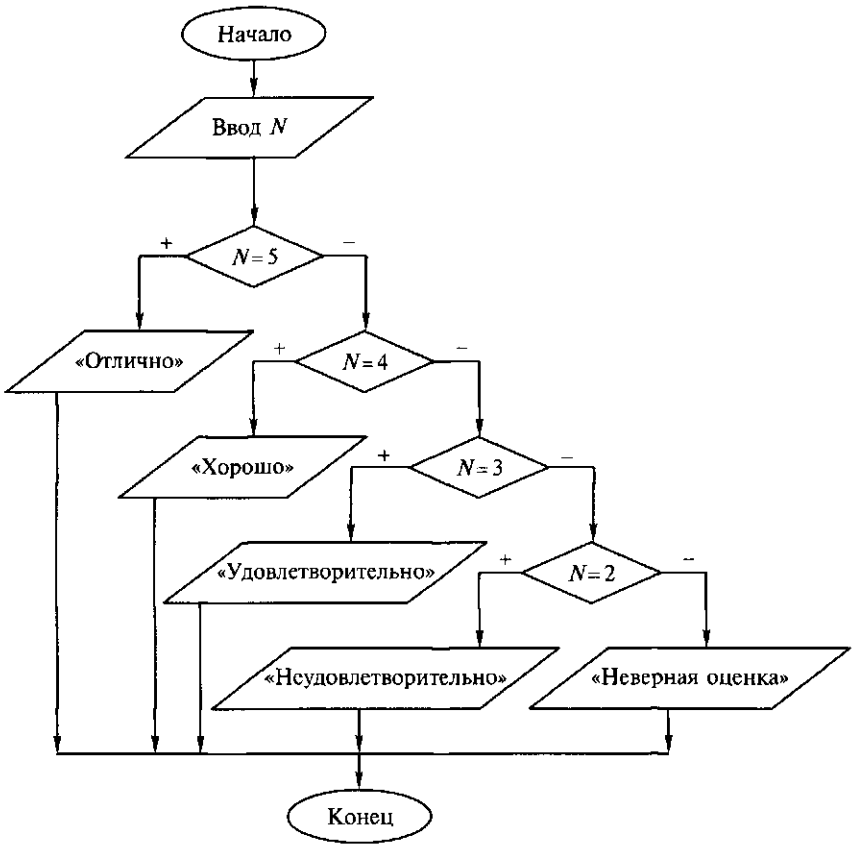


Рис. 2.14. Блок-схема алгоритма перевода пятибалльной оценки в ее наименование

```
            3: WriteLn('Удовлетворительно');
            2: WriteLn('Неудовлетворительно')
    Else WriteLn('Нет такой оценки')
End;
```

Формат оператора выбора описывается синтаксической диаграммой, показанной на рис. 2.15, где <Селектор> — это выражение любого порядкового типа, <Константа> — постоянная величина того же типа, что и селектор, а <Оператор> — любой простой или составной оператор.

Выполнение оператора выбора происходит следующим образом: вычисляется выражение — селектор, затем в списках кон-

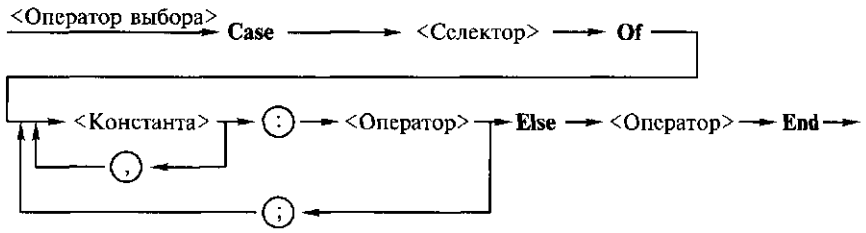


Рис. 2.15. Синтаксическая диаграмма оператора выбора

стант находится значение, совпадающее с полученным значением селектора, а далее исполняется оператор, помеченный данной константой. Если такой константы не найдено, происходит переход к выполнению оператора, следующего после **Else**.

Покажем на примере использование списка констант в операторе выбора. Следующая программа сообщает, сдал студент экзамен или не сдал, т. е. если он получил оценку 3, 4 или 5, то экзамен сдан, а если — 2, то экзамен не сдан:

```
Case N Of
     3, 4, 5 : WriteLn('Экзамен сдан')
           2 : WriteLn('Экзамен не сдан')
     Else WriteLn('Нет такой оценки');
```

Так же, как условный оператор, оператор выбора может использоваться в неполной форме, т. е. без ветви **Else**.

2.11. ПРОГРАММИРОВАНИЕ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ

Существует три типа циклических структур: цикл с предусловием, цикл с постусловием и цикл по параметру.

Цикл с предусловием. Рассмотрим синтаксическую диаграмму оператора цикла «Пока», или цикла с предусловием (рис. 2.16). Здесь сначала вычисляется <Логическое выражение>. Пока его значение равно True, выполняется <Оператор> — тело цикла. При этом <Оператор> может быть как простым, так и составным.

Для примера приведем фрагмент программы на Паскале, вычисляющий с заданной точностью ε сумму гармонического ряда

$$1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{I}+\cdots$$

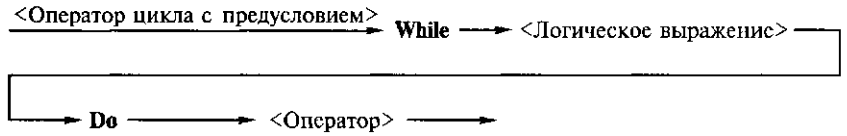


Рис. 2.16. Синтаксическая диаграмма цикла с предусловием

Суммирование прекращается, когда очередное слагаемое становится меньше ε или когда целая переменная *I* достигает значения MaxInt:

```
S := 0;
I := 1;
While (1/I >= Eps) And (I < MaxInt) Do
Begin
  S := S + 1/I;
  I := I + 1
End;
```

Цикл с постусловием. Синтаксическая диаграмма оператора Цикла «До», или цикла с постусловием, приведена на рис. 2.17.

Исполнение данного цикла повторяется до того момента, когда <Логическое выражение> станет равным True.

Предыдущая задача с использованием цикла с постусловием решается следующим образом:

```
S := 0;
I := 1;
Repeat
  S := S + 1/I;
  I := I + 1
Until (1/I < Eps) Or (I >= MaxInt);
```

Цикл по параметру. Рассмотрим задачу вычисления суммы целых чисел от *M* до *N* прямым суммированием. Данную задачу можно записать в виде

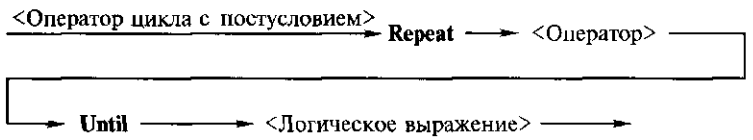


Рис. 2.17. Синтаксическая диаграмма цикла с постусловием

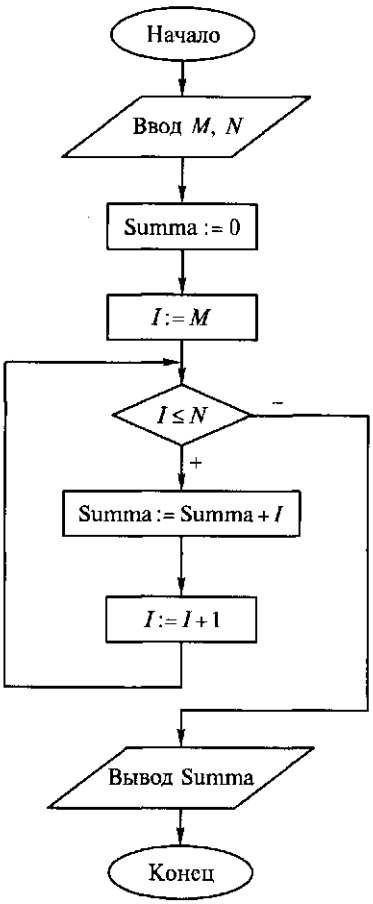


Рис. 2.18. Блок-схема алгоритма суммирования целых чисел

$$\text{Summa} = \begin{cases} \sum_{I=M}^{N} I, & \text{если } M \le N; \\ 0, & \text{если } M > N. \end{cases}$$

Блок-схема алгоритма решения этой задачи приведена на рис. 2.18, а соответствующую программу с использованием структуры цикла «Пока» можно записать следующим образом:

```
Program Adding;
Var I, M, N, Summa : Integer;
```

```
Begin
  Write('M =');
  ReadLn(M);
  Write('N =');
  ReadLn(N);
  Summa := 0;
  I := M;
  While I <= N Do
  Begin
      Summa := Summa + I;
      I := Succ(I)
  End;
  WriteLn('Сумма равна ', Summa)
End.
```

Теперь введем новый тип циклической структуры, которая называется «цикл по параметру». Блок-схема алгоритма суммирования с использованием этой структуры приведена на рис. 2.19, а

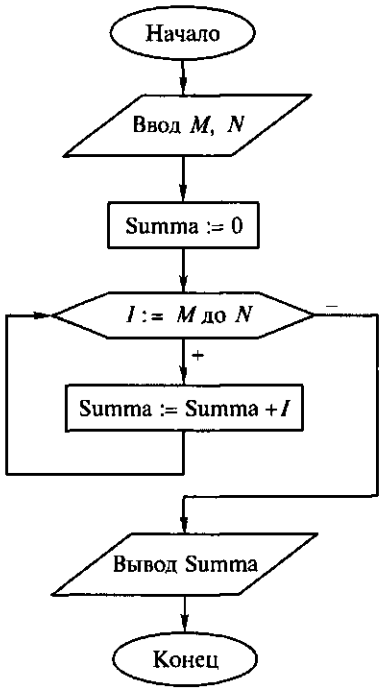


Рис. 2.19. Блок-схема алгоритма суммирования с циклом по параметру

программу на Паскале для решения данной задачи можно записать следующим образом:

```
Program Summering_2;
Var I, M, N, Summa: Integer;
Begin  Write('M =');
       ReadLn(M);
       Write('N =');
       ReadLn(N);
       Summa := 0;
       For I := M To N Do
           Summa := Summa + I;
       WriteLn('Сумма равна ', Summa)
End.
```

Здесь целая переменная *I* принимает последовательность всех значений в диапазоне от *M* до *N*. При каждом значении *I* выполняется тело цикла. После последнего выполнения цикла при *I* = *N* происходит выход из цикла на продолжение алгоритма. Цикл выполнится хотя бы один раз, если $M \le N$, и не выполнится ни разу при $M > N$.

В приведенной программе используется оператор цикла по параметру, синтаксическая диаграмма которого показана на рис. 2.20, где

```
<параметр цикла> ::= <имя простой переменной
порядкового типа>
```

Выполнение оператора **For** (в первом варианте **To**) происходит по следующей схеме.

1. Вычисляются значения <Выражение 1> и <Выражение 2>, причем только один раз при входе в цикл.

2. Параметру цикла присваивается значение <Выражение 1>.

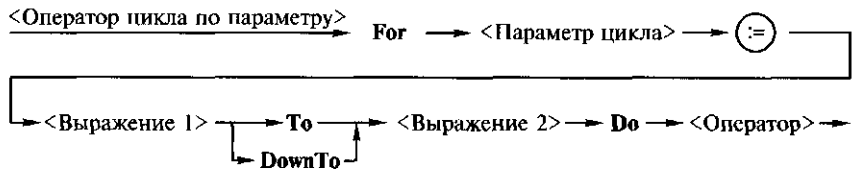


Рис. 2.20. Синтаксическая диаграмма цикла с параметром

3. Значение параметра цикла сравнивается с значением <Выражение 2>. Если параметр цикла меньше или равен этому значению, то выполняется тело цикла, в противном случае выполнение цикла заканчивается.

4. Значение параметра цикла изменяется на следующее значение в его типе (для целых чисел — увеличивается на единицу) и происходит возврат к п. 3 данной схемы.

Оператор цикла **For** объединяет в себе действия, которые при использовании цикла **While** выполняют различные операторы: присваивание параметру начального значения, сравнение с конечным значением, изменение значения на следующее.

Как известно, результат суммирования целых чисел не зависит от порядка суммирования. Например, в рассматриваемой задаче числа можно складывать и в обратном порядке, т.е. от N до M ($N \geq M$). Для этого можно использовать второй вариант оператора цикла **For**:

```
Summa := 0;
For I := N DownTo M Do
Summa := Summa + I;
```

DownTo буквально переводится «вниз до». В данном случае параметр цикла изменяется по убыванию, т.е. при каждом повторении цикла параметр изменяет свое значение на предыдущее (равносильно $i := \mathrm{pred}(i)$). Следовательно, цикл не выполнится ни разу, если $N < M$.

Работая с оператором **For**, необходимо учитывать следующие правила:

- параметр цикла не может иметь тип real;
- в теле цикла нельзя изменять переменную — параметр цикла;
- при выходе из цикла значение переменной — параметра цикла — является неопределенным.

В следующем примере в качестве параметра цикла **For** используем символьную переменную. Пусть требуется получить на экране десятичные коды букв латинского алфавита. Как известно, латинские буквы в таблице кодировки упорядочены по алфавиту. Фрагмент соответствующей программы можно записать следующим образом:

```
For C := 'a' To 'z' Do
Write (C, ' - ', Ord(C));
```

Здесь переменная C типа char.

Теперь самостоятельно запрограммируйте вывод кодировки латинского алфавита в обратном порядке (от 'z' до 'a')?

УПРАЖНЕНИЯ

1. Составить на Паскале программу полного решения биквадратного уравнения.

2. Используя оператор выбора, составить программу, которая по введенному номеру месяца в году будет выводить соответствующее время года (зима, весна, лето, осень).

3. Используя операторы цикла While, Repeat и For, составить три варианта программы вычисления $N!$.

4. Составить программу, по которой последовательность символов будет вводиться до тех пор, пока не встретится строчная или прописная латинская буква z. Подсчитать, сколько раз среди вводимых символов встретится буква W.

5. Вычислить сумму квадратов всех целых чисел, попадающих в интервал $(\ln x; e^x)$, где $x > 1$.

6. Вычислить число точек с целочисленными координатами, попадающих в круг с радиусом R ($R > 0$) и с центром в начале координат.

7. Напечатать таблицу значений функции $\sin x$ и $\cos x$ в интервале $[0; 1]$ с шагом 0.1 в следующем виде:

x	$\sin x$	$\cos x$
0.0000	0.0000	1.0000
0.1000	0.0998	0.9950
...	...	...
1.0000	0.8415	0.5403

8. Напечатать в возрастающем порядке все трехзначные числа, в десятичной записи которых нет одинаковых цифр.

9. Дано целое $n > 2$. Напечатать все простые числа из интервала $[2; n]$.

2.12. ПОДПРОГРАММЫ

Понятие вспомогательного алгоритма уже рассматривалось в подразд. 1.5. В языках программирования вспомогательные алгоритмы называются *подпрограммами*. В Паскале различают два вида подпрограмм: *процедуры* и *функции*.

Рассмотрим следующий пример: даны два натуральных числа a и b. Требуется определить наибольший общий делитель трех величин: $a + b$, $|a - b|$, $a \cdot b$. Запишем это в виде НОД($a + b$, $|a - b|$, $a \cdot b$).

Идея решения данной задачи состоит в следующем математическом факте: если x, y, z — три натуральных числа, то НОД(x, y, z) = = НОД(НОД(x, y), z). Иначе говоря, сначала следует найти НОД двух величин, а затем НОД полученного значения и третьего числа (попробуйте это доказать).

Очевидно, что вспомогательным алгоритмом для решения поставленной задачи является алгоритм получения наибольшего общего делителя двух чисел. Следовательно, эта задача решается с помощью известного алгоритма Евклида (см. подразд. 1.3), который запишем в форме процедуры на алгоритмическом языке:

```
процедура Евклид(цел M, N, K);
нач
  пока M <> N
  нц
       если M > N
       то M := M-N
       иначе N := N-M
       кв
  кц;
  K := M
кон
```

Здесь M и N являются формальными параметрами процедуры, т.е. это параметры-аргументы, а K — параметр-результат.

Основной алгоритм решения исходной задачи следующий:

```
алг задача
цел a, b, c
нач ввод(a, b)
    Евклид(a + b, |a-b|, c)
    Евклид(c, a * b, c)
    вывод(c)
кон
```

Процедуры. В отличие от А Я, где процедура является внешней по отношению к вызывающей программе, процедуры в Паскале описываются в разделе описания подпрограмм. Программа решения исходной поставленной задачи на Турбо Паскале будет иметь следующий вид:

```
Program NOD1;
Var A, B, C : Integer;
Procedure Evklid(M, N : Integer; Var K : Integer);
Begin
  While M <> N Do
  If M > N
  Then M := M-N
  Else N := N-M;
  K := M
End;
Begin
  Write('A =');
  ReadLn(A);
  Write('B =');
  ReadLn(B);
  Evklid(A + B, Abs(A-B), C);
  Evklid(C, A * B, C);
  WriteLn('НОД =', C)
End.
```

В данном случае обмен аргументами и результатами между основной программой и процедурой производится через параметры (формальные и фактические). Существует и другой механизм обмена — через глобальные переменные, что будет рассмотрено далее.

Синтаксическая диаграмма описания процедуры показана на рис. 2.21.

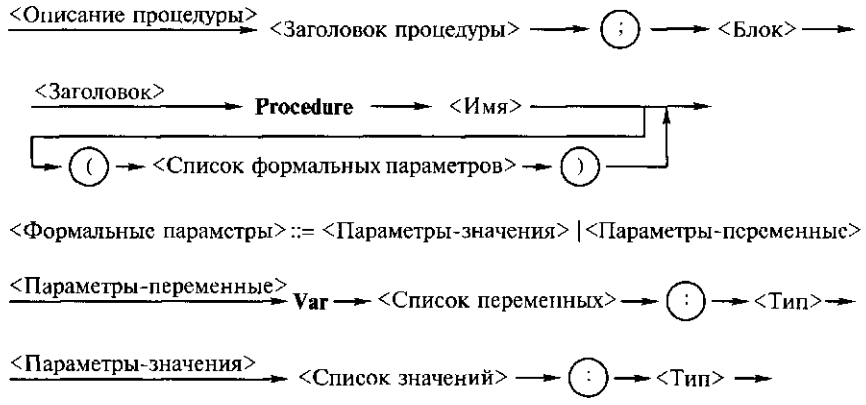


Рис. 2.21. Синтаксическая диаграмма описания процедуры

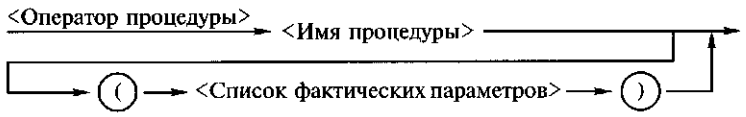


Рис. 2.22. Структура оператора обращения к процедуре

Из диаграммы видно, что процедура может иметь параметры, а может и не иметь. Чаще всего аргументы представляются как параметры-значения (но они могут быть и параметрами-переменными), для передачи результатов используются параметры-переменные.

Процедура в качестве результата может передавать в вызывающую программу множество значений (в частном случае — одно), а может и не передавать ни одного значения. Теперь рассмотрим правила обращения к процедуре. Обращение к процедуре производится в форме оператора процедуры (рис. 2.22).

Если описана процедура с формальными параметрами, то и обращение к ней производится оператором процедуры с фактическими параметрами. Правила соответствия между формальными и фактическими параметрами следующие: соответствие *по количеству*, *по последовательности* и *по типам*.

Первый вариант взаимодействия формальных и фактических параметров называется *передачей по значению*: вычисляется значение фактического параметра (выражения) и это значение присваивается соответствующему формальному параметру. Второй вариант взаимодействия формальных и фактических параметров называется *передачей по имени*: при выполнении процедуры имя формальной переменной заменяется именем соответствующей фактической переменной (в откомпилированной программе имени переменной соответствует адрес ячейки памяти).

В рассмотренном примере формальные параметры M и N являются параметрами-значениями. Это аргументы процедуры, при обращении к которой первый раз им соответствуют значения выражений $A + B$ и $Abs(A - B)$, а второй раз — значения C и $A \cdot B$. Параметр K является параметром-переменной, в которой получают результат работы процедуры. В обоих обращениях к процедуре соответствующим фактическим параметром является переменная C, через которую основная программа получает результат.

Теперь рассмотрим пример программы, решающей исходную поставленную задачу с использованием процедуры без параметров:

```
Program NOD2;
Var A, B, K, M, N : Integer;
Procedure Evklid;
Begin
  While M <> N Do
  If M > N
  Then M := M-N
  Else N := N-M;
  K := M
End;
Begin
  Write('A =');
  ReadLn(A);
  Write('B =');
  ReadLn(B);
  M := A + B;
  N := Abs(A-B);
  Evklid;
  M := K;
  N := A * B;
  Evklid;
  WriteLn('НОД равен ', K)
End.
```

Чтобы разобраться в этом примере, требуется рассмотреть понятие *область действия описания*.

Областью действия описания любого программного объекта (переменной, типа, константы и т.д.) является тот блок, в котором расположено это описание. Если данный блок вложен в другой (подпрограмма), то присутствующие в нем описания являются *локальными* и действуют только в пределах внутреннего блока. Описания же, стоящие во внешнем блоке, являются *глобальными* по отношению к внутреннему блоку. Если глобально описанный объект используется во внутреннем блоке, то на него распространяется внешнее (глобальное) описание.

В программе NOD1 переменные *M*, *N*, *K* — локальные внутри процедуры, а переменные *A*, *B*, *C* — глобальные. Однако внутри процедуры переменные *A*, *B*, *C* не используются. Связь между внешним блоком и процедурой осуществляется через параметры.

В программе NOD2 все переменные являются глобальными. В процедуре Evklid нет ни одной локальной переменной (нет и параметров), поэтому используемые в ней переменные *M* и *N* получают свои значения через оператор присваивания в основном

блоке программы. Результат получают в глобальной переменной *K*, значение которой выводится на экран.

Использование механизма передачи значений через параметры делает процедуру более универсальной, независимой от основной программы. Однако в некоторых случаях оказывается удобнее использовать передачу значений через глобальные переменные, особенно для процедур, работающих с большими объемами информации. В этой ситуации глобальное взаимодействие экономит память ЭВМ.

Функции. Теперь выясним, что представляет собой подпрограмма-функция. Обычно функция используется в том случае, если результатом подпрограммы должна быть скалярная (простая) величина. Тип результата называется типом функции. В Турбо Паскале допускаются функции строкового типа. Синтаксическая диаграмма описания функции представлена на рис. 2.23.

Как и у процедуры, у функции в списке формальных параметров могут иметься параметры-переменные и параметры-значения, которые являются ее аргументами. При этом если аргументы передаются глобально, параметры в списке могут отсутствовать.

Программа решения исходной поставленной задачи с использованием функции будет иметь следующий вид:

```
Program NOD3;
Var A, B, Rez: Integer;
Function Evklid(M, N : Integer) : Integer;
Begin
  While M <> N Do
  If M > N
  Then M := M-N
  Else N := N-M;
  Evklid := M
End;
Begin
  Write('A = ');
```

Рис. 2.23. Синтаксическая диаграмма описания функции

```
  ReadLn(A);
  Write('B = ');
  ReadLn(B);
  Rez := Evklid(Evklid(A + B, Abs(A-B)), A * B);
  WriteLn('NOD равен ', Rez)
End.
```

Из данного примера видно, что в отличие от процедуры в теле функции *результат присваивается переменной с именем функции.*

Обращение к функции является операндом в выражении и записывается в следующей форме:

```
<имя функции> (<список фактических параметров>)
```

Правила соответствия между формальными и фактическими параметрами у функции те же, что и у процедуры. Сравнивая рассмотренные программы, можно сделать вывод, что программа NOD3 имеет определенные преимущества перед другими. Функция позволяет получить результат посредством выполнения одного оператора присваивания. Здесь же показано, что фактическим аргументом при обращении к функции может быть эта же функция.

По правилам стандартного Паскаля возврат в вызывающую программу из подпрограммы происходит, когда выполнение подпрограммы доходит до конца (последний End). Однако в Турбо Паскале имеется средство, позволяющее выйти из подпрограммы в любом ее месте. Это оператор-процедура Exit. Например, функцию определения наибольшего из двух заданных вещественных чисел можно описать следующим образом:

```
Function Max(X, Y : Real): Real;
Begin
  Max := X;
  If X > Y Then Exit Else Max := Y
End;
```

Еще раз об области действия описаний. В Паскале неукоснительно действует следующее правило: *любой программный объект (константа, переменная, тип и др.) должен быть описан перед использованием в программе.* Иначе говоря, описание объекта должно предшествовать его первому появлению в других фрагментах программы. Это правило относится и к подпрограммам.

Структура взаимного расположения описаний подпрограмм в некоторой условной программе схематически показана на рис. 2.24.

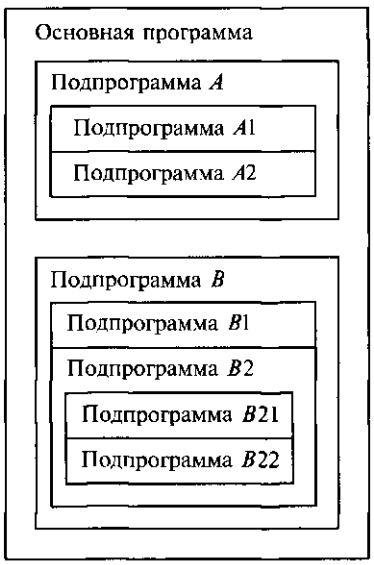


Рис. 2.24. Пример структуры программы

Попробуем, используя эту схему, разобраться в вопросе об области действия описаний подпрограмм.

Любая подпрограмма может использоваться лишь в пределах области действия ее описания. Например, область действия подпрограмм *A* и *B* — основная программа, поэтому из основной программы можно обратиться к подпрограммам *A* и *B*. В свою очередь, в подпрограмме *B* могут быть обращения к подпрограмме *A*, однако из *A* нельзя обратиться к *B*, поскольку описание *A* предшествует описанию *B*. Подпрограммы *A*1 и *A*2 локализованы в подпрограмме *A* и могут использоваться только в ней, причем из *A*2 можно обратиться к *A*1, а из *A*1 в *A*2 нельзя.

Из подпрограммы *B*1 можно обратиться к подпрограмме *A*, поскольку ее описание является глобальным по отношению к *B*1, но нельзя обратиться к *A*1, поскольку область действия описания *A*1 не распространяется на блок подпрограммы *B*.

Из подпрограммы *B*22 можно обратиться только к подпрограммам *B*21, *B*1 и *A*. (Определите самостоятельно, почему.)

Если одно и то же имя описано во внешнем блоке (глобально) и внутреннем (локально), то последнее описание (локальное) перекрывает первое в пределах внутреннего блока. Рассмотрим две программы:

```
Program Example1;
Var X : Integer;
Procedure P;
Var X: Integer;
Begin WriteLn('X = ', X)
End;
Begin X := 1;
  P
End.
```

```
Program Example2;
Var X : Integer;
Procedure P;
Begin
  WriteLn('X = ', X)
End;
Begin X := 1;
  P
End.
```

В результате работы первая программа выдаст результат *X* = ..., где на месте отточия будет какое-то произвольное значение, соответствующее неопределенной величине *x*.

Вторая программа выдаст результат *X* = 1.

В первой полпрограмме переменная с именем *X* описана и глобально, и локально, но процедура выводит значение локальной переменной, которой ничего не присвоено. Здесь идентификатором *X* обозначены две совершенно разные величины, которым соответствуют две разные ячейки памяти.

Во второй программе переменная *X* одна на всю программу и описана глобально, поэтому значение 1, присвоенное ей в основной программе, передается и в подпрограмму.

УПРАЖНЕНИЯ

1. Составить два варианта программы вычисления площади кольца по значениям внутреннего и внешнего радиусов: с процедурой и с функцией, используя подпрограмму вычисления площади круга.
2. По координатам вершин треугольника вычислить его периметр, используя подпрограмму вычисления длины отрезка между двумя точками.
3. Даны три целых числа. Определить, используя подпрограмму, у которого из них больше сумма цифр.
4. Определить площадь выпуклого четырехугольника по заданным координатам вершин, используя подпрограмму-функцию вычисления длины отрезка и подпрограмму-процедуру вычисления площади треугольника по формуле Герона.

2.13. ВЫЧИСЛЕНИЕ РЕКУРРЕНТНЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Рекуррентная последовательность. Понятие рекуррентной последовательности известно из курса математики. Вводится оно

следующим образом: пусть известно k чисел: $a_1, ..., a_k$, которые являются началом числовой последовательности. Следующие элементы этой последовательности вычисляются по формулам:

$$a_{k+1} = F(a_1, ..., a_k); \; a_{k+2} = F(a_2, ..., a_{k+1});$$

$$a_{k+3} = F(a_3, ..., a_{k+2}) ...,$$

где $F(...)$ — функция от k аргументов.

Формула вида

$$a_i = F(a_{i-1}, a_{i-2}, ..., a_{i-k})$$

называется *рекуррентной*, а величина k — *глубиной рекурсии*.

Другими словами, *рекуррентная последовательность — это бесконечный ряд чисел, каждое из которых, за исключением k начальных, вычисляется через предыдущие*.

Примерами рекуррентных последовательностей являются соответственно арифметическая и геометрическая прогрессии:

$$a_1 = 1, \; a_2 = 3, \; a_3 = 5, \; a_4 = 7, \; a_5 = 9, \; ...; \qquad (2.1)$$

$$a_1 = 1, \; a_2 = 2, \; a_3 = 4, \; a_4 = 8, \; a_5 = 16, \; ... \qquad (2.2)$$

Рекуррентная формула данной арифметической прогрессии

$$a_i = a_{i-1} + 2.$$

Рекуррентная формула данной геометрической прогрессии

$$a_i = 2a_{i-1}.$$

Глубина рекурсии в обоих случаях равна единице (такую зависимость также называют одношаговой рекурсией). В целом рекуррентная последовательность описывается совокупностью начальных значений и рекуррентной формулой. Все это можно объединить в одну ветвящуюся формулу, которая для арифметической прогрессии будет иметь вид

$$a_i = \begin{cases} 1, & \text{если } i = 1; \\ a_{i-1} + 2, & \text{если } i > 1, \end{cases}$$

а для геометрической —

$$a_i = \begin{cases} 1, & \text{если } i = 1; \\ 2a_{i-1}, & \text{если } i > 1. \end{cases}$$

Числовая последовательность вида

1; 1; 2; 3; 5; 8; 13; 21; 34; 55 ...,

известная в математике под названием чисел Фибоначчи, в которой, начиная с третьего элемента, каждое число равно сумме значений двух предыдущих, является рекуррентной последовательностью с глубиной, равной двум (двухшаговая рекурсия). Опишем ее в ветвящейся форме:

$$f_i = \begin{cases} 1, & i = 1,\ i = 2; \\ f_{i-1} + f_{i-2}, & i > 2. \end{cases}$$

Введение представления о рекуррентных последовательностях позволяет по новому взглянуть на некоторые уже известные нам задачи. Например, факториал от целого числа $n!$ можно рассматривать как значение n-го элемента следующего ряда чисел:

$$a_0 = 1;\ a_1 = 1!;\ a_2 = 2!;\ a_3 = 3!;\ a_4 = 4!;\ \ldots$$

Рекуррентное описание такой последовательности будет иметь вид

$$a_i = \begin{cases} 1, & i = 0; \\ a_{i-1} i, & i > 0. \end{cases}$$

Программирование вычислений рекуррентных последовательностей. С рекуррентными последовательностями связаны следующие задачи:

1) вычисление заданного (n-го) элемента последовательности;

2) математическая обработка определенной части последовательности (например, вычисление суммы или произведения первых n членов);

3) подсчет числа элементов на заданном отрезке последовательности, удовлетворяющих определенным свойствам;

4) определение номера первого элемента, удовлетворяющего определенному требованию;

5) вычисление и сохранение в памяти заданного числа элементов последовательности.

Данный перечень, не претендуя на полноту, охватывает наиболее часто встречающиеся типы задач. В задачах 1...4 не требуется одновременно хранить в памяти множество элементов числового ряда: *его элементы могут получаться последовательно в одной переменной, сменяя друг друга.*

Пример 2.4. Требуется вычислить n-й элемент арифметической прогрессии (2.1).

Программа, решающая данную задачу, следующая:

```
Var N, I: 0..MaxInt;
  A : Real;
Begin
  Write('N = ');
  ReadLn(N);
  A := 1;
  For  I := 2 To N Do
       A := A + 2;
  WriteLn('A(', N:1, ')= ', A:6:0)
End.
```

Рекуррентная формула $a_i = a_{i-1} + 2$ перешла в оператор $A := A + 2$.

Пример 2.5. Требуется просуммировать первые n элементов геометрической прогрессии (2.2), не используя формулу для суммы прогрессии.

Программа решения данной задачи следующая:

```
Var N, I: 0..MaxInt;
  A, S : Real;
Begin
  Write('N = '); ReadLn(N);
  A := 1;
  S := A;
  For I := 2 To N Do
  Begin
      A := 2 * A;
      S := S + A
  End;
  WriteLn('Сумма равна ', S:6:0)
End.
```

При вычислении рекуррентной последовательности с глубиной, равной двум, уже нельзя обойтись одной переменной, что видно из следующего примера.

Пример 2.6. Требуется вывести на печать первые $n (n \geq 3)$ чисел Фибоначчи и подсчитать, сколько среди них четных чисел.

Программа решения данной задачи следующая:

```
Var N, I, K, F, F1, F2 : 0..MaxInt;
Begin
  F1 := 1; F2 := 1;
  K := 0;
  WriteLn('F(1) = ', F1, ' F(2) = ', F2);
  For I := 3 To N Do
  Begin
       F := F1 + F2;
```

```
        WriteLn('F(', I : 1, ') = ', F);
        If Not Odd(F) Then K := K + 1;
        F1 := F2; F2 := F
    End;
    WriteLn('Количество четных чисел в последователь-
    сти равно ', K)
End.
```

Для последовательного вычисления двухшаговой рекурсии требуются три переменные, поскольку для вычисления очередного элемента необходимо помнить значения двух предыдущих.

Пример 2.7. Для заданного вещественного x и малой величины ε (например, $\varepsilon = 0{,}000001$) требуется вычислить сумму ряда

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots,$$

включив в нее только слагаемые, превышающие ε.

Из математики известно, что сумма такого бесконечного ряда имеет конечное значение, равное e^x, где $e = 2{,}71828\ldots$ — основание натурального логарифма. Поскольку элементы этого ряда представляют собой убывающую последовательность чисел, стремящуюся к нулю, суммирование следует производить до первого слагаемого, по абсолютному значению не превышающего ε.

Если слагаемые в этом выражении обозначить

$$a_0 = 1;\ a_1 = x;\ a_2 = \frac{x^2}{2!};\ a_3 = \frac{x^3}{3!},\ \ldots,$$

то обобщенная формула для i-го элемента будет следующей:

$$a_i = \frac{x^i}{i!}.$$

Нетрудно увидеть, что между элементами данной последовательности имеется рекуррентная зависимость. Ее можно найти интуитивно, но можно и вывести формально. Правда, для этого требуется догадаться, что рекурсия — одношаговая, и что каждый следующий элемент получается посредством умножения предыдущего на некоторый множитель, т.е.

$$a_i = Ka_{i-1}.$$

Используя обобщенную формулу, запишем:

$$\frac{x^i}{i!} = K\frac{x^{i-1}}{(i-1)!},$$

откуда

$$K = \frac{x^i/i!}{x^{i-1}/(i-1)!} = \frac{x}{i}.$$

Действительно,

$$a_0 = 1;\ a_1 = a_0 \frac{x}{2};\ a_3 = a_2 \frac{x}{3}\ \ldots$$

Следовательно, данная рекуррентная последовательность может быть описана следующим образом:

$$a_i = \begin{cases} 1, & \text{если } i = 0; \\ a_{i-1} \frac{x}{i}, & \text{если } i > 0. \end{cases}$$

И, наконец, приведем программу, решающую поставленную задачу:

```
Var A, X, S, Eps : Real;
    I : Integer;
Begin
  Write('X = '); ReadLn(X);
  Write('Epsilon = '); ReadLn(Eps);
  A := 1; S := 0; I := 0;
  While Abs(A) > Eps Do
  Begin
    S := S + A;
    I := I + 1;
    A := A * X/I
  End;
  WriteLn('Сумма ряда равна ', S : 10 : 4)
End.
```

Значения одношаговой рекуррентной последовательности здесь, как и прежде, вычисляются в одной переменной.

Каждое повторное выполнение цикла в этой программе приближает значение S к искомому (уточняет значащие цифры в его записи). Такой вычислительный процесс в математике называется *итерационным процессом*. Соответственно циклы, реализующие итерационный вычислительный процесс, называются *итерационными циклами*. Для их организации используются операторы While или Repeat.

Пример 2.8. Для заданного натурального числа N и вещественного x ($x > 0$) требуется вычислить значение выражения

$$\sqrt{x + \sqrt{x + \ldots + \sqrt{x}}}.$$

В этом случае рекуррентность не столь очевидна. Попробуем найти ее методом индукции. Будем считать, что искомое выражение есть N-й элемент последовательности вида

$$a_1 = \sqrt{x};\ a_2 = \sqrt{x + \sqrt{x}};\ a_3 = \sqrt{x + \sqrt{x + \sqrt{x}}}\ \ldots,$$

откуда видна следующая связь:

$$a_2 = \sqrt{x + a_1};\ a_3 = \sqrt{x + a_2};\ \ldots;\ a_i = \sqrt{x + a_{i-1}}.$$

Теперь поставленную задачу решить очень просто:

```
Var A, X : Real; I, N : Integer;
Begin
  Write('X = '); ReadLn(X);
  Write('N = '); ReadLn(N);
  A := Sqrt(X);
  For  I := 2 To N Do
       A := Sqrt(X + A);
  WriteLn('Ответ: ', A)
End.
```

Однако к решению всех приведенных задач можно подойти и иначе, используя возможность рекурсивоного определения функции в Паскале.

Подробно рекурсивные подпрограммы Паскаля рассматриваются в подразд. 2.17. Из описания арифметической прогрессии в форме рекуррентной последовательности непосредственно следует способ определения функции для вычисления заданного элемента.

Сделаем это для общего случая, определив арифметическую прогрессию с начальным значением a_0 и разностью d:

$$a_i = \begin{cases} a_0, & \text{если } i = 1; \\ a_{i-1} + d, & \text{если } i > 1. \end{cases}$$

Соответствующая подпрограмма-функция будет иметь вид

```
Function Progres(A0, D : Real; I : Integer): Real;
Begin
  If I = 1
  Then Progres := A0
  Else Progres := Progres(A0, D, I - 1) + D
End;
```

Следующая программа выводит на экран первые 20 чисел Фибоначчи, значения которых вычисляет рекурсивная функция Fibon:

```
Var K : Byte;
Function Fibon(N : Integer) : Integer;
Begin
  If (N = 1) Or (N = 2)
  Then Fibon := 1
  Else Fibon := Fibon(N-1) + Fibon(N-2)
End;
Begin
  For K := 1 To 20 Do WriteLn(Fibon(K))
End.
```

Необходимо отметить, что использование рекурсивных функций ведет к замедлению счета. Кроме того, можно столкнуться с проблемой нехватки длины стека, в котором запоминается «маршрут» рекурсивных обращений.

Рекуррентные последовательности часто используются для решения разного рода эволюционных задач, т.е. задач, в которых прослеживается какой-то процесс, развивающийся во времени. Рассмотрим такую задачу.

Пример 2.9. В ходе лечебного голодания масса тела пациента за 30 дней снизилась с 96 до 70 кг. Требуется вычислить, чему была равна масса тела пациента через k дней после начала голодания для $k = 1, 2, ..., 29$.

Обозначим массу тела пациента в i-й день p_i ($i = 0, 1, 2,..., 30$). Из условия задачи известно, что $p_0 = 96$ кг, $p_{30} = 70$ кг. Пусть K — коэффициент пропорциональности убывания массы за один день. Тогда

$$p_1 = p_0 - Kp_0 = (1 - K)\ p_0;\ p_2 = (1 - K)p_1;$$

$$p_3 = (1 - K)p_2;\ ...;\ p_i = (1 - K)\ p_i - 1,$$

т.е. имеем последовательность, описываемую следующей рекуррентной формулой:

$$p_i = \begin{cases} 96, & \text{если } i = 0; \\ (1 - K)p_{i-1}, & \text{если } 1 \le i \le 30. \end{cases}$$

Коэффициент K можно найти, используя условие $p_{30} = 70$ и выполнив «обратные» подстановки:

$$p_{30} = (1 - K)p_{29} = (1 - K)^2 p_{28} = (1 - K)^3 p_{27} = ... =$$
$$= (1 \cdot K)^{30} p_0 = (1 - K)^{30} \cdot 96.$$

Из равенства $(1 - K)^{30} \cdot 96 = 70$ находим:

$$1 - K = \left(\frac{70}{96}\right)^{\frac{1}{30}}.$$

Дальнейшее программирование становится очевидным:

```
Var  I : Byte;
     P, Q : Real;
Begin
  P := 96;
  Q := Exp(1/30 * Ln(70/96));
  For I := 1 To 29 Do
  Begin
       P := Q * P;
       WriteLn(I, '-й день - ', P : 5 : 3, ' кг')
  End
End.
```

УПРАЖНЕНИЯ

1. Рекуррентная последовательность определена следующим образом:

$$a_i = \begin{cases} 1, & \text{если } i = 0; \\ a_{i-1} i + \dfrac{1}{i}, & \text{если } i > 0. \end{cases}$$

Для заданного натурального n получить значение a_n.

2. Дана последовательность

$$a_i = \begin{cases} 1, & \text{если } i = 0,\ i = 1; \\ a_{i-2} + \dfrac{a_{i-1}}{i-1}, & \text{если } i > 1. \end{cases}$$

Вычислить произведение элементов с 1-го по 20-й.

3. Используя рекуррентный подход, вычислить сумму многочлена 10-й степени по формуле Горнера:

$$10x^{10} + 9x^9 + 8x^8 + \ldots + 2x^2 + x = ((((((((((10x + 9)x + 8)x + \ldots + 2)x + 1)x.$$

где x — заданное вещественное число.

4. Для заданных вещественного x и натурального N вычислить следующую цепную дробь:

$$x/(1 + x/(2 + x/(3 + x/(\ldots /(N + x))\ldots).$$

5. Вычислить и вывести все члены числового ряда

$$1; \frac{1}{2!}; \frac{1}{3!}; \ldots; \frac{1}{N!},$$

превышающие значение 10^{-5}.

6. Функцию $y = \sqrt{x}$ можно вычислить как предельное значение последовательности, которую определяют следующей рекуррентной формулой:

$$y_k = \frac{1}{2}\left(y_{k-1} + \frac{x}{y_{k-1}}\right) \text{ для } k = 1, 2, \ldots$$

Здесь начальное значение y_0 задается произвольно (желательно ближе к $\sqrt{x}$). За приближенное с точностью ε значение корня берется первое y_k, для которого выполняется условие $|y_k - y_k - 1| < \varepsilon$. Составить соответствующую программу вычисления.

2.14. ГРАФИЧЕСКИЕ СРЕДСТВА ТУРБО ПАСКАЛЯ

До сих пор мы использовали экран компьютера только для вывода символьной информации — чисел, текстов. Однако Турбо Паскаль позволяет выводить на экран рисунки, чертежи, графики функций, диаграммы, т.е. все то, что принято называть компьютерной графикой.

В стандартном Паскале графический вывод не предусмотрен. Однако на разных типах компьютеров, в разных реализациях Паскаля существуют различные программные средства графического вывода: специальные наборы данных, функций, процедур. При этом имеются общие понятия и средства, свойственные любому варианту реализации графики в любом языке программирования. Рассмотрим такие базовые средства.

Начиная с четвертой версии Турбо Паскаля для IBM PC появилась мощная графическая библиотека, организованная в модуль Graph. В приложении 2 в справочной форме дано описание основных компонентов этого модуля. Для подключения модуля Graph в начале программы необходимо написать строку

```
Uses Graph;
```

Графические режимы экрана. Для вывода графических изображений необходимо перевести экран в один из графических режимов. В графическом режиме можно с помощью программы управлять состоянием каждого пиксела — точечного элемента экрана.

Графические режимы отличаются:

- размером графической сетки ($M \times N$, где M — число точек по горизонтали; N — число точек по вертикали);
- цветностью (числом воспроизводимых на экране цветов).

Допустимые режимы зависят от типа монитора и соответствующего графического драйвера, используемого на компьютере.

Для установки графического режима экрана существуют соответствующие процедуры. В модуле Graph процедура установки графического режима экрана имеет следующий заголовок:

```
Procedure InitGraph(Var Driver, Mode : Integer;
Path : String);
```

Здесь целая переменная Driver определяет тип графического драйвера, целая переменная Mode задает режим работы графического драйвера, а Path — выражение типа String, содержащее маршрут поиска файла графического драйвера.

Списки констант модуля Graph, определяющих типы драйверов и режимы, даны в приложении 2 (см. табл. П2.1 и П2.2).

Приведем пример программы, инициализирующей графический режим VGAHi для работы с драйвером VGA (монитор типа VGA):

```
Uses Graph;
Var Driver, Mode : Integer;
Begin
  Driver : = VGA;{Драйвер}
  Mode : = VGAHi;{Режим работы}
  InitGraph(Driver, Mode, 'C:\TP\BGI');
```

Здесь указывается, что файл egavga.bgi с драйвером для VGA-монитора находится в каталоге C:\TP\BGI. Режим VGAHi соответствует графической сетке 640 × 480 с палитрой из 16 цветов.

Возможны также автоматические определение типа драйвера и установка режима, что позволяет программе работать с разными типами мониторов, не внося изменений в текст:

```
Driver := Detect;
InitGraph(Driver, Mode, 'C:\TP\BGI');
```

При этом автоматически устанавливается режим с наибольшими разрешающей способностью и цветностью. После окончания работы в графическом режиме следует вернуться в текстовый режим экрана.

В модуле Graph процедура возвращения в текстовый режим имеет заголовок

```
Procedure CloseGraph;
```

Цвет фона и цвет рисунка. На цветном мониторе можно изменять окраску экрана. Установленная окраска экрана называется цветом фона. Рисунок на этом фоне наносится с помощью разнообразных линий: прямых, окружностей, прямоугольников, ломаных и др. Цвета этих линий также могут меняться.

Имена констант, определяющих 16 цветов палитры для мониторов типа EGA, VGA, приведены в табл. П2.3 приложения.

Заголовок процедуры установки цвета фона

```
Procedure SetBkColor(Color : Word);
```

Здесь Color — выражение целого типа, определяющее номер цвета фона.

Заголовок процедуры установки цвета линий

```
Procedure SetColor(Color : Word);
```

Заметим, что если в качестве номера цвета линии указан 0, то это всегда совпадает с цветом фона (невидимая линия).

При необходимости очистить графический экран (стереть рисунок) используется процедура очистки экрана, имеющая следующий заголовок:

```
Procedure ClearDevice;
```

В результате выполнения этой процедуры экран заполняется установленным цветом фона.

Графические координаты. Положение каждого пиксела графической сетки однозначно определяется указанием его координат. Расположение на экране графических осей координат показано на рис. 2.25. Горизонтальная ось *X* направлена слева направо, вертикальная ось *Y* — сверху вниз.

Предельные графические координаты, соответствующие режиму VGAHi, указаны на рис. 2.25.

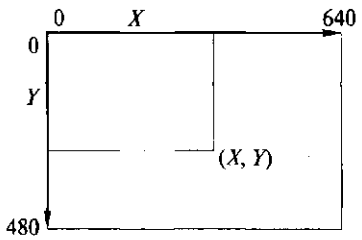


Рис. 2.25. Система экранных координат

Можно определить максимальные координаты по осям, соответствующие данному драйверу, с помощью следующих двух целочисленных функций:

```
Function GetMaxX;
Function GetMaxY;
```

Графическое окно. Область вывода изображения может быть ограничена любым прямоугольником в пределах экрана. Такая область называется графическим окном. Существует процедура, устанавливающая положение графического окна на экране.

Заголовок процедуры назначения графического окна

```
Procedure SetViewPort(X1, Y1, X2, Y2 : Integer;
Clip : Boolean);
```

Здесь (*X*1, *Y*1) — координаты левого верхнего угла окна; (*X*2, *Y*2) — координаты правого нижнего угла окна; Clip — ограничитель фигур.

Если Clip = True, то все построения производятся только в пределах окна, в противном случае они могут выходить за его пределы.

После установки окна координаты точек внутри него отсчитываются от его верхнего левого угла.

Существует понятие графического курсора, который в отличие от символьного курсора на экране не виден. Графический курсор указывает на текущую позицию на экране. При входе в графический режим координаты текущей позиции (0, 0).

Процедура назначения координат графического курсора:

```
Procedure MoveTo(X, Y : Integer);
```

Здесь *X*, *Y* — устанавливаемые координаты курсора, которые указываются относительно левого верхнего угла окна или (если окно не установлено) экрана.

Процедура *поставить точку* — основная процедура получения изображения, поскольку любой рисунок складывается из точек. Состояние светящейся точки определяется ее координатами на экране и цветом.

Заголовок процедуры выставления точки на графическом экране

```
Procedure PutPixel(X, Y : Integer; Color : Word);
```

Здесь *X*, *Y* — координаты точки; Color — цвет точки.

Пример 2.10. Программа, устанавливающая по центру экрана графическое окно размером 100 × 100, заливающая его желтым фоном и заполняющая синими точками, расположенными через четыре позиции, будет иметь следующий вид:

```
Uses Graph;
Var  Driver, Mode : Integer;
     X, Y, X1, Y1, X2, Y2, Xc, Yc : Integer;
Begin
{----Инициализация графического режима----}
     Driver : = Detect;
     InitGraph(Driver, Mode, 'C:\TP\BGI');
{----Определение координат центра экрана----}
     Xc := GetMaxX Div 2;
     Yc := GetMaxY Div 2;
{----Определение координат графического окна----}
     X1 := Xc - 50;
     Y1 := Yc - 50;
     X2 := Xc + 50;
     Y2 := Yc + 50;
{----Установка графического окна----}
     SetViewPort(X1, Y1, X2, Y2, True);
{----Установка цвета фона и очистка экрана----}
     SetBkColor(Yellow);
     ClearDevice;
{----Расстановка точек в окне----}
     For X := 1 To 25 Do
       For Y := 1 To 25 Do
       PutPixel(4 * X, 4 * Y, Blue);
{----Задержка изображения на экране до нажатия
<Enter>----}
     ReadLn;
{----Выход из графического режима в символьный----}
     CloseGraph;
End.
```

Графические примитивы. Любое изображение можно построить из точек, однако программировать получение сложного рисунка или чертежа, используя только процедуру *поставить точку*, было бы слишком неудобно и громоздко. В любом графическом пакете существуют процедуры рисования основных геометрических фигур: прямых линий, окружностей, эллипсов, прямоугольников и т.д. Такие фигуры называют *графическими примитивами*.

Рассмотрим несколько основных процедур рисования графических примитивов, имеющихся в модуле Graph.

- Линия с заданными координатами концов (*X*1, *Y*1) и (*X*2, *Y*2):

```
Procedure Line(X1, Y1, X2, Y2 : Integer);
```

- Линия от текущей точки до точки с координатами (*X*, *Y*):

```
Procedure LineTo(X, Y : Integer);
```

- Линия от текущей точки до точки с заданными приращениями координат (*DX*, *DY*):

```
Procedure LineRel(DX, DY : Integer);
```

- Прямоугольник с заданными координатами верхнего левого угла (*X*1, *Y*1) и нижнего правого угла (*X*2, *Y*2):

```
Procedure Rectangle(X1, Y1, X2, Y2 : Integer);
```

Окружность с центром в точке (*X*, *Y*) и с радиусом *R* в пикселах:

```
Procedure Circle(X, Y : Integer; R : Word);
```

- Дуга окружности с центром в точке (*X*, *Y*), с радиусом *R*, с начальным углом Beg*A* и конечным углом End*A* (углы измеряются в градусах против часовой стрелки от направления оси *X*):

```
Procedure Arc(X, Y : Integer; BegA, EndA, R :
Word);
```

- Эллипсная дуга с центром в точке (*X*, *Y*), с начальным и конечным углами соответственно Beg*A* и End*A*, с горизонтальным радиусом RX и вертикальным радиусом *RY*:

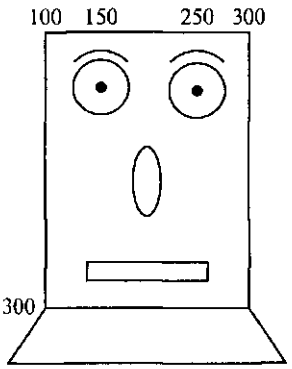
```
Procedure Ellipse(X, Y : Integer; BegA, EndA, RX,
RY : Word);
```

Пример 2.11. Составим программу, рисующую голову робота (рис. 2.26).

Рисунок содержит два прямоугольника, две окружности, две дуги, эллипс, три прямые линии и две черные точки. Заранее определив все координаты и размеры элементов рисунка, запишем программу:

```
Uses Graph;
Var Driver, Mode : Integer;
Begin
{----Инициализация графического режима----}
  Driver := Detect;
```

Рис. 2.26. Чертеж головы Робота




```
  InitGraph(Driver, Mode, 'C:\TP\BGI');
  SetColor(White); {Белый цвет рисунка}
  SetBkColor(Black); {Черный цвет фона}
  Rectangle(100, 100, 300, 300); {Голова}
  Circle(150, 170, 30); {Левый глаз}
  Circle(250, 170, 30); {Правый глаз}
  Arc(150, 170, 45, 135, 40); {Левая бровь}
  Arc(250, 170, 45, 135, 40); {Правая бровь}
  Ellipse(200, 250, 0, 359, 10, 20); {Нос}
  Rectangle(130, 280, 270, 290); {Рот}
  MoveTo(100, 300); {Установка вниз влево}
  LineTo(50, 350); {Три}
  LineTo(350, 350); {Линии}
  LineTo(300, 300); {Шеи}
  PutPixel(150, 170, Black); {Левый зрачок}
  PutPixel(250, 170, Black); {Правый зрачок}
  ReadLn; {Задержка}
  CloseGraph; {Выход из графики}
End.
```

В приведенном примере все линии сплошные нормальной толщины. Модуль Graph позволяет управлять стилем линии (сплошная, пунктирная, точечная и др.) и ее толщиной, для чего существует процедура SetLineStile (см. табл. П2.4) приложения.

Закраски и заполнения. Среди графических примитивов существуют закрашенные области. Цвет закраски определяется процедурой SetColor. Кроме того, можно управлять рисунком закраски (типом заполнения). Это может быть сплошная закраска, заполнение редкими точками, крестиками, штрихами и т.д. Константы для указания типа заполнения даны в табл. П2.5 приложения.

Процедура определения типа заполнения (Fill) и цвета (Color) имеет следующий заголовок:

```
Procedure SetFillStyle(Fill, Color : Word);
```

Процедура заполнения прямоугольной области с заданными координатами углов:

```
Procedure Bar(X1, Y1, X2, Y2 : Integer);
```

Процедура обведения линией (SetLineColor, SetLineStyle) и закраски (SetFillStyle) эллипса:

```
Procedure FillEllips(X, Y, RX, RY : Integer);
```

Процедура обведения линией и закраски эллипсного сектора:

```
Procedure Sector(X, Y : Integer; BegA, EndA, RX, RY
: Word);
```

Процедура обведения линией и закраски сектора окружности:

```
Procedure PieSlice(X, Y : Integer; BegA, EndA :
Word);
```

Наконец, можно закрасить любую область, ограниченную замкнутой линией. Для этого нужно указать какую-нибудь точку внутри этой области (*X*, *Y*) и цвет граничной линии (Border). Соответствующая процедура имеет вид

```
Procedure FloodFill(X, Y : Integer; Border : Word);
```

Модуль Graph позволяет выводить на графический экран тексты. Не будем детально обсуждать эту проблему (необходимую информацию можно найти в специальной литературе), приведем лишь пример одной текстовой процедуры, с помощью которой выводится в графическое окно символьная строка (Txt), начиная с указанной позиции (*X*, *Y*):

```
Procedure OutTextXY(X, Y : Integer; Txt : String);
```

Например, чтобы вывести под рис. 2.26 строку ЭТО РОБОТ, следует в программу добавить оператор

```
OutTextXY(195, 400, 'ЭТО РОБОТ');
```

Построение графика функции. Одно из приложений компьютерной графики обеспечивает наглядное представление результа-

тов математических расчетов. Нарисованный график функции, начерченные диаграммы, линии уровней распределения пространственных зависимостей и другие делают результаты расчетов обозримее, нагляднее, понятнее.

Рассмотрим сейчас лишь один простейший вариант математической графики — построение графика функции.

Решение задачи построения на экране дисплея графика функции $y = F(x)$ удобно проводить в следующем порядке.

1. Определить границы значений аргумента, в пределах которых будет строиться график, обозначив их X_{min} (нижняя граница) и X_{max} (верхняя граница).

2. Для данной области значений аргумента определить предельные значения функции: Y_{min} и Y_{max}. Эти значения необязательно должны быть точными. Они могут быть оценочными.

3. Задать границы графического окна, в пределах которого будет рисоваться график: $[Xg_{min}, Xg_{max}]$, $[Yg_{min}, Yg_{max}]$. Поскольку в графических координатах вертикальная ось направлена вниз, то $Yg_{min} > Yg_{max}$.

Таким образом, имеются две системы координат: (X, Y) — математические координаты (в литературе чаще используется термин «мировые координаты») и (Xg, Yg) — графические координаты.

Нетрудно получить формулы, связывающие графические и математические координаты:

$$Xg = Xg_{min} + \left[\frac{Xg_{max} - Xg_{min}}{X_{max} - X_{min}}(X - X_{min})\right];$$

$$Yg = Yg_{min} + \left[\frac{Yg_{max} - Yg_{min}}{Y_{max} - Y_{min}}(Y - Y_{min})\right].$$

Здесь квадратные скобки означают округление до целого значения (функция Round).

Построение графика функции может производиться либо точечным методом, либо кусочно-линейным. При *точечном методе* график строится как последовательность точек, расположенных максимально близко, т.е. производится «попикселный» перебор значений аргумента в интервале $[Xg_{min}, Xg_{max}]$ с выставлением точек с соответствующими Y-координатами.

При *кусочно-линейном методе* задается математический шаг ΔX, рассчитывается последовательность значений (X_i, Y_i):

$$X_i = X_{max} + i\Delta X;\ Y_i = F(X_i);$$

$$i = 0, 1, \ldots, n;\quad n = \frac{X_{max} - X_{min}}{\Delta X},$$

и график строится в виде отрезков прямых, проведенных через точки (X_i, Y_i), $(X_i + 1, Y_{i+1})$.

Пример 2.12. Требуется составить программу построения графика функции $y = \sin x$ для $x \in [0; 2\pi]$, используя точечный метод.

Из условия задачи следует, что $X_{min} = 0$, $X_{max} = 2\pi$. В этих пределах функция $\sin x$ изменяется от -1 до 1, поэтому $Y_{min} = -1$, $Y_{max} = 1$.

Выберем следующие границы графического окна: $Xg_{min} = 10$; $Xg_{max} = 200$; $Yg_{min} = 140$; $Yg_{max} = 40$.

График строится в виде последовательности точек со следующими математическими координатами:

$$X_i = X_{min} + ih;\ Y_i = \sin(X_i);\ i = 0, \ldots, 190.$$

Шаг h выбирается минимально возможным, соответствующим шагу графической сетки:

$$h = \frac{X_{max} - X_{min}}{Xg_{max} - Xg_{min}} = \frac{2\pi}{190} = \frac{\pi}{95}.$$

Приведенные формулы пересчета математических координат в графические примут следующий вид:

$$Xg = 10 + \left[\frac{200 - 10}{2\pi} X\right] = 10 + \left[\frac{95}{\pi}\right];$$

$$Yg = 140 + \left[\frac{40 - 140}{2}(Y + 1)\right] = 90 - [50Y].$$

Вместе с графиком функции строятся оси координат. Ось X имеет координату $Yg = 90$, а ось Y — координату $Xg = 10$.

Программа построения графика следующая:

```
Uses Graph;
Var Driver, Mode : Integer;
    X : Real; Xg, Yg, I : Integer;
Begin
{---- Инициализация графического режима----}
    Driver := Detect;
    InitGraph(Driver, Mode, 'C:\TP\BGI');
    SetColor(White); {Белый цвет линий}
    SetBkColor(Black); {Черный цвет фона}
    Line(10, 90, 200, 90); {Ось X}
    Line(10, 20, 10, 160); {Ось Y}
{---- Построение графика функции желтыми точкам---}
    X := 0;
    For I := 0 To 190 Do
    Begin Xg := 10 + Round(95/Pi * X);
      Yg := 90 - Round(50 * Sin(X));
      PutPixel(Xg, Yg, Yellow);
```

```
      X := X + Pi/95
    End;
{----Разметка осей, запись функции----}
    OutTextXY(15, 30, 'Y');
    OutTextXY(205, 90, 'X');
    OutTextXY(130, 40, 'Y = Sin(X)');
    Readln; {Задержка}
    CloseGraph; {Выход из графики}
End.
```

УПРАЖНЕНИЯ

1. Составить программу «Звездное небо»: расстановки в черном окне случайным образом белых точек, работа которой заканчивается нажатием клавиши.

2. Изменить программу «Звездное небо» таким образом, чтобы наряду с зажиганием новых звезд происходило угасание (закрашивание цветом фона) уже светящихся звезд.

3. В программу из примера 2.11 внести изменения, в результате которых Робот окажется раскрашенным в разные цвета.

4. Используя линии и другие графические примитивы, составить программу, рисующую дом.

5. Составить программу рисования на экране шахматного поля.

6. Написать универсальную процедуру построения графика функции $y = F(x)$ точечным методом. Процедура должна иметь следующие параметры: X_{min}, X_{max}, Y_{min}, Y_{max}, Xg_{min}, Xg_{max}, Yg_{min}, Yg_{max}. Функция $F(x)$ описывается во внешней подпрограмме-функции.

7. Исследовав область определения и выбрав расположение координатных осей, построить на экране, используя процедуру из предыдущей задачи, графики следующих функций:

а) $y = \frac{1}{x+1}$; б) $y = \frac{x+3}{x-2}$; в) $y = 1 + \frac{2}{x} + \frac{3}{x^2}$.

2.15. СИМВОЛЬНЫЕ СТРОКИ

Рассмотрим тип данных, относящийся к числу структурированных — строковый (строка). Следует заметить, что строковый тип данных есть в Турбо Паскале и отсутствует в стандартном Паскале.

Строка — это последовательность символов. Каждый символ занимает 1 байт памяти (код ASCII). Количество символов в строке называется ее длиной. Длина строки может находиться в диапазоне от 0 до 255. Строковые величины могут быть константами и переменными.

Строковая константа — это последовательность символов, заключенная в апострофы. Например:

```
'Язык программирования ПАСКАЛЬ'
'IBM PC - computer',
'33-45-12'.
```

Строковая переменная описывается в разделе описания переменных следующим образом:

```
Var <идентификатор> : String[<максимальная длина
строки>].
```

Например:

```
Var Name : String[20].
```

Длина строки может и не указываться в описании. В этом случае подразумевается, что она максимальна — 255 символов. Например:

```
Var Slovo : String.
```

Строковая переменная занимает в памяти на 1 байт больше, чем указанная в описании ее длина. Дело в том, что один (нулевой) байт содержит значение *текущей длины строки*. Если строковой переменной не присвоено никакого значения, ее текущая длина равна нулю.

По мере заполнения строки символами ее текущая длина возрастает, но она не должна превышать максимального по описанию значения.

Символы внутри строки индексируются (нумеруются), начиная с единицы. Каждый отдельный символ идентифицируется именем строки с индексом, заключенным в квадратные скобки. Например:

```
Name[5], Name[i], Slovo[k+1].
```

Индекс может быть положительной константой, переменной величиной и выражением целого типа. Значение индекса не должно выходить за границы описания.

Тип String и стандартный тип Char совместимы. Строки и символы могут употребляться в одних и тех же выражениях.

Строковые выражения строятся из строковых констант, переменных, функций и знаков операций. Над строковыми данными допустимы операции сцепления и отношения.

Операция сцепления (+) применяется для соединения нескольких строк в одну результирующую строку. Сцеплять можно как строковые константы, так и переменные. Например:

```
'ЭВМ'+' IBM'+' PC'.
```

В результате получится строка

'ЭВМ IBM PC'.

Длина результирующей строки не должна превышать 255 символов.

Операции отношения (=, <, >, <=, >=, <>) производят сравнение двух строк, в результате чего получают логическую величину (True или False). Операция отношения имеет более низкий приоритет, чем операция сцепления. Сравнение строк производится слева направо до первого несовпадающего символа, и большей считается та строка, в которой первый несовпадающий символ имеет больший номер в таблице символьной кодировки.

Если строки имеют различную длину, но в общей части символы совпадают, считается, что более короткая строка меньше, чем более длинная. Строки равны, если они полностью совпадают по длине и содержат одни и те же символы. Например:

Выражение	Результат
'cosm1' < 'cosm2'	True
'pascal' > 'PASCAL'	True
'Ключ_' <> 'Ключ'	True
'MS DOS' = 'MS DOS'	True

- Функция Copy(S, Poz, N) — выделяет из строки *S* подстроку длиной *N* символов, начиная с позиции Poz. Здесь *N* и Poz — целочисленные выражения. Например:

Значение *S*	Выражение	Результат
'ABCDEFG'	Copy(S, 2, 3)	'BCD'
'ABCDEFG'	Copy(S, 4, 4)	'DEFG'

- Функция Concat (*S*1, *S*2, ..., *SN*) — выполняет сцепление (конкатенацию) строк *S*1, ..., *SN* в одну строку. Например:

Выражение	Результат
Concat('AA', 'XX', 'Y')	'AAXXY'

- Функция Length(S) — определяет текущую длину строки *S*. Результатом будет являться значение целого типа. Например:

Значение *S*	Выражение	Результат
'test-5'	Length(S)	6
'(A+B)*C'	Length(S)	7

- Функция Pos(S1, S2) — обнаруживает первое появление в строке *S*2 подстроки *S*1. В результате получают целое число, равное номеру позиции, где находится первый символ подстроки *S*1. Если в *S*2 подстроки *S*1 не обнаружено, результат равен 0. Например:

Значение *S*2	Выражение	Результат
'abcdef'	Pos('cd', S2)	3
'abcdcdef'	Pos('cd', S2)	3
'abcdef'	Pos('k', S2)	0

- Процедура Delete(S, Poz, N) — удаляет *N* символов из строки *S*, начиная с позиции Poz. Например:

Исходное значение *S*	Оператор	Конечное значение
'abcdefg'	Delete(S,3,2)	'abefg'
'abcdefg'	Delete(S,2,6)	'a'

В результате выполнения процедуры уменьшается текущая длина строки в переменной *S*.

- Процедура Insert(*S*1, *S*2, Poz) — вставляет строку *S*1 в строку *S*2, начиная с позиции Poz. Например:

Начальное *S*2	Оператор	Конечное *S*2
'ЭВМ PC'	Insert('IBM-',S2,5)	'ЭВМ IBM-PC'
'Рис. 2'	Insert('N',S2,6)	'Рис. N2'

Пример 2.13. Программа получения из слова «ВЕЛИЧИНА» слова «НАЛИЧИЕ»:

```
Program Slovo_1;
Var S11, S12 : String[10];
Begin
  S11 := 'ВЕЛИЧИНА';
  S12 := Copy(S11,7,2) + Copy(S11,3,4) + S11[2];
  WriteLn(S12)
End.
```

Пример 2.14. Программа получения из слова «СТРОКА» слова «СЕТКА»:

```
Program Slovo_2;
Var S1 : String[10];
Begin
  S1 := 'СТРОКА';
  Delete(S1,3,2);
  Insert('Е',S1,2);
  WriteLn(S1)
End.
```

Пример 2.15. Программа, формирующая символьную строку, состоящую из N звездочек (где N — целое число, $1 \le N \le 255$):

```
Program Stars;
Var A : String;
  N, I : Byte;
Begin
  Write ('Введите число звездочек');
  ReadLn(N);
  A := '';
  For  I := 1 To N Do
       A := A + '*';
  WriteLn(A)
End.
```

Здесь строковой переменной A сначала присваивается значение пустой строки (' '), а затем к ней присоединяются звездочки.

Пример 2.16. Программа подсчета в символьной строке числа цифр, предшествующих первому символу «!»:

```
Program C;
Var S : String;
    K, I : Byte;
Begin
    WriteLn('Введите строку');
    ReadLn(S);
    K := 0;
    I := 1;
    While (I <= Length(S)) And (S[I] <> '!') Do
    Begin
      If (S[I] >= '0') And (S[i] <= '9')
      Then      K := K + 1;
      I := I + 1
    End;
    WriteLn ('Количество цифр до символа "!" равно ', K)
End.
```

В этой программе переменная *K* играет роль счетчика цифр, а переменная *I* — роль параметра цикла. Выполнение цикла закончится при первом же выходе на символ «!» или (если в строке такого символа нет) при выходе на конец строки. Символ *S*[*I*] является цифрой, если истинно отношение '0' ≤ *S*[*I*] ≤ '9'.

Пример 2.17. Дана символьная строка вида: '*A* ⊕ *B* ='. Здесь *A* и *B* — десятичные цифры; знаком ⊕ обозначен один из знаков операций (+, –, *). Иначе говоря, дано арифметическое выражение с двумя однозначными числами и знаком равенства, например: '5 + 7 ='. Требуется, чтобы машина вычислила это выражение и после знака «=» вывела результат. Операция деления не рассматривается, чтобы иметь дело только с целыми числами.

Программа решения такой задачи называется *интерпретатором*. Интерпретатор должен расшифровать содержание строки и выполнить соответствующую арифметическую операцию. От исходной символьной информации он должен перейти к работе с числовой информацией.

Если предположить, что строка составлена только из четырех символов в соответствии с указанным форматом, то программа решения будет довольно простой:

```
Program Interpretator;
Var Str : String[4];
  A, B : 0..9;
  C : -100..100;
Begin
{-----Ввод исходной строки----}
  WriteLn('Введите выражение!');
  WriteLn;
  Read(Str);
{----Преобразование цифровых символов в числа----}
  A := Ord(Str[1]) - Ord('0');
  B := Ord(Str[3]) - Ord('0');
{----Выполнение арифметической операции----}
  Case Str[2] Of
       '+' : C := A + B;
       '-' : C := A - B;
       '*' : C := A * B
  End;
{----Вывод результата----}
  WriteLn(C:2)
End.
```

В данной программе роль селектора в операторе выбора играет символьная величина Str[2]. Если она равна «+», выполним оператор *C* := *A* + *B*; если она равна «-», выполним оператор *C* := *A* – *B*; если она равна «*», выполним оператор *C* := *A* * *B*. Для любых других значений Str[2] не выполним ни один из операторов присваивания, и значение переменной *C* остается неопределенным.

В этой программе использована неполная форма оператора Case, т.е. без ветви Else.

Программа выдачи сообщения о получении неверного символа в Str[2] будет следующая:

```
Case Str[2] Of
   '+' : C := A + B;
   '-' : C := A-B;
   '*' : C := A * B
Else WriteLn('Неверный знак операции')
End;
```

Разберемся теперь в том, как будет выполняться программа Interpretator.

После ввода строки цифровые символы переводятся в соответствующие десятичные числа. Затем интерпретируется знак операции. В зависимости от этой интерпретации выполняется одно из трех арифметических действий. Далее результат выводится на экран после символа «=».

УПРАЖНЕНИЯ

1. Составить программу получения из слова «дисковод» слова «воск», используя операцию сцепления и функцию Copy.
2. Составить программу получения слова «правило» из слова «операция», используя процедуры Delete и Insert.
3. В заданном слове заменить первый и последний символы на символ «*».
4. В заданном слове произвести обмен первого и последнего символов.
5. К заданному слову присоединить столько символов «!», сколько в нем имеется букв (например, из строки «УРА» получить «УРА!!!»).
6. В заданной строке вставить пробел после каждого символа.
7. Удвоить каждую букву введенного слова.
8. Перевернуть введенную строку (например, из строки «ДИСК» получить «КСИД»).
9. В заданной строке удалить все пробелы.
10. Составить программу перевода строки, представляющей собой запись целого числа, в соответствующую величину целого типа.

2.16. МАССИВЫ

В повседневной и научной практике часто приходится встречаться с информацией, представленной в табличной форме. Сред-

Таблица 2.7. Линейная таблица температур

Месяц года	1	2	3	4	5	6	7	8	9	10	11	12
Температура, °С	–21	–18	–7,5	5,6	10	18	22,2	24	17	5,4	–7	–18

немесячные значения температур за определенный год содержит табл. 2.7. Такая таблица, представляющая собой последовательность упорядоченных чисел, называется *линейной*. Если требуется какая-то математическая обработка этих данных, то для их обозначения обычно вводят индексную символику. Например, T_1 обозначают температуру января (первого месяца), а T_5 — мая и т.д. В общем виде множество значений, содержащихся в таблице, обозначается следующим образом:

$$\{Ti\},\ i = 1, \ldots, 12.$$

Порядковые номера элементов (1, 15, i и др.) называются *индексами*. Индексированные величины удобнее использовать при записи их для математической обработки. Например, для вычисления среднегодовой температуры используется формула

$$T_{\text{ср}} = \frac{1}{12}\sum_{i=1}^{12} T_i.$$

Теперь представьте, что требуется собрать информацию о среднемесячных температурах за 10 лет, например с 1981 по 1990 г. Используем для этого *прямоугольную таблицу* (табл. 2.8), в которой столбцы соответствуют годам, а строки — месяцам.

Таблица 2.8. Прямоугольная таблица температур

Год	Месяц года											
	1	2	3	4	5	6	7	8	9	10	11	12
1981	–23	–17	–8,4	6,5	14	18,6	25	19	12,3	5,6	–4,5	–19
1982	–16	–8,5	–3,2	7,1	8,4	13,8	28,5	21	6,5	2	–13	–20
1983	–9,8	–14	–9,2	4,6	15,6	21	17,8	20	11,2	8,1	–16	–21
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1990	–25	–9,7	–3,8	8,5	13,9	17,8	23,5	17,5	10	5,7	–14	–20

Удачно выбранная форма записи данных обеспечивает удобство и быстроту получения требуемой информации. Например, из табл. 2.8 легко узнать, в каком году был самый холодный январь или в каком месяце был самым нестабильный температурный режим за указанное десятилетие.

Для значений, хранящихся в подобной таблице, удобно использовать двухиндексные обозначения. Например, $H_{1981,2}$ — обозначить температуру в феврале 1981 г. и т.д. При этом всю совокупность данных, составляющих таблицу, математически можно записать следующим образом:

$$\{H_{i,j}\};\ i = 1981 \ldots 1990;\ j = 1,\ \ldots,\ 12.$$

Обработка таких данных производится с использованием двухиндексных величин. Например, средняя температура марта за 10 лет

$$H_{\text{ср.март}} = \frac{1}{10}\sum_{i=1981}^{1990} H_{i,3},$$

а средняя температура за весь десятилетний период

$$H_{\text{ср}} = \frac{1}{10}\sum_{i=1981}^{1990}\left[\frac{1}{12}\sum_{j=1}^{12} H_{i,j}\right] = \frac{1}{120}\sum_{i=1981}^{1990}\sum_{j-1}^{12} H_{i,j}.$$

В Паскале аналогом таблиц является структурированный тип данных, который называется *регулярным типом*, или *массивом*. Как и таблица, массив представляет собой совокупность пронумерованных однотипных значений, имеющих общее имя. Элементы массива обозначаются как переменные с индексами. Индексы записываются в квадратных скобках после имени массива. Например:

```
T[1], T[5], T[i], H[1981, 9], H[i, j]
```

Массив, хранящий линейную таблицу, называется *одномерным*, а прямоугольную — *двухмерным*. В программах могут использоваться массивы и большей размерности.

Тип элементов массива является его *базовым типом*. Очевидно, для рассмотренных массивов температур базовым типом является вещественный (Real).

Описание массивов. Переменная регулярного типа описывается в разделе описания переменных в следующей форме:

```
Var <идентификатор> : Array [<тип индекса>] Of <тип
компонент>
```

Чаще всего в качестве типа индекса употребляется интервальный. Например, рассмотренный ранее одномерный массив среднемесячных температур можно описать следующим образом:

```
Var T : Array [1..12] Of Real;
```

Описание массива определяет его размещение в памяти и правила дальнейшего употребления в программе. Последовательные элементы массива располагаются в последовательных ячейках памяти (*T*[1], *T*[2] и т.д.), причем значения индекса не должны выходить из диапазона 1...12. В качестве индекса может употребляться любое выражение соответствующего типа. Например, *T*[*i* + *j*], *T*[*m* div 2].

Тип индекса может быть любым скалярным порядковым типом, кроме Integer. Например, в программе могут присутствовать следующие описания:

```
Var  Cod : Array[Char] Of 1..100;
     L : Array[Boolean] Of Char;
```

В данной программе допустимы следующие обозначения элементов массивов:

```
Cod['x']; L[True]; Cod[Chr(65)]; L[a > 0].
```

В некоторых случаях бывает удобно в качестве индекса использовать перечисляемый тип данных. Например, данные о количестве учеников в четырех десятых классах одной школы могут храниться в следующем массиве:

```
Type Index = (A, B, C, D);
Var Class_10 : Array[Index] Of Byte;
```

И если, например, элемент Class_10[A] равен 35, это означает, что в 10 А классе 35 человек. Такое индексирование делает программу более наглядной.

Часто структурированному типу данных присваивается имя в разделе типов, которое затем используется в разделе описания переменных:

```
Type  Mas1 = Array [1..100] Of Integer;
      Mas2 = Array [-10..10] Of Char;
Var   Num : Mas1; Sim : Mas2;
```

До сих пор речь шла об одномерных массивах, в которых типы элементов скалярные.

Многомерный массив в Паскале трактуется как одномерный массив, тип элементов которого также является массивом (*массив массивов*). Например, рассмотренную ранее прямоугольную таблицу можно хранить в массиве, описанном следующим образом:

```
Var H : Array[1981..1990] Of Array[1..12] Of Real;
```

Приведем примеры обозначения некоторых элементов этого массива:

```
H[1981][1]; H[1985][10]; H[1990][12].
```

Однако чаще употребляется следующая, эквивалентная форма обозначения элементов двухмерного массива:

```
H[1981, 1]; H[1985, 10]; H[1990, 12].
```

Здесь переменная *H*[1981] обозначает всю первую строку таблицы, т.е. весь массив температур за 1981 г.

Эквивалентным вариантом приведенного описания двухмерного массива является также следующее:

```
Type  Month = Array [1..12] Of Real;
      Year = Array [1981..1990] Of Month;
Var   H : Year;
```

Наиболее краткий вариант описания данного массива имеет вид

```
Var H : Array[1981..1990, 1..12] Of Real;
```

Аналогично трехмерный массив можно определить как одномерный массив, у которого элементами являются двухмерные массивы. Например:

```
Var A : Array[1..10, 1..20, 1..30] Of Integer;
```

Это описание массива, состоящего из $10 \cdot 20 \cdot 30 = 6\,000$ целых чисел и занимающего в памяти $6\,000 \cdot 2 = 12\,000$ байт. В Паскале нет ограничения сверху на размерность массива. Однако в каждой конкретной реализации Паскаля ограничивается объем памя-

ти, выделяемый под массивы. В Турбо Паскале это ограничение составляет 64 Кбайт.

По аналогии с математикой одномерные числовые массивы часто называют *векторами*, а двухмерные — *матрицами*.

В Паскале не допускается употребление динамических массивов, т.е. массивов, размер которых определяется в процессе выполнения программы. Изменение размеров массива происходит через изменение в тексте программы и повторную компиляцию. Для упрощения таких изменений определяют индексные параметры в разделе констант:

```
Const Imax = 10; Jmax = 20;
Var Mas : Array[1..Imax, 1..Jmax] Of Integer;
```

Теперь для изменения размеров массива Mas и всех операторов программы, связанных с этими размерами, достаточно отредактировать только одну строку в программе — раздел констант.

Действия над массивом как единым целым. Такие действия допустимы лишь в двух случаях:

- присваивание значений одного массива другому;
- в операциях отношений «равно», «не равно».

В обоих случаях массивы должны иметь одинаковые типы (тип индексов и тип элементов). Например:

```
Var P, Q : Array[1..5,1..10] Of Real;
```

При выполнении операции присваивания

```
P := Q
```

все элементы массива *P* станут равны соответствующим элементам массива *Q*.

Как уже отмечалось, в многомерных массивах переменная с индексом может обозначать целый массив. Например, если в таблице *H* требуется данные за 1989 г. сделать такими же, как за 1981 г. (девятой строке присвоить значение первой строки), то это можно сделать следующим образом:

```
H[1989] := H[1981].
```

Поменять местами значения этих строк в таблице можно через третью переменную того же типа:

```
P := H[1989]; H[1989] := H[1981]; H[1981] := P;
```

Здесь *P* описана следующим образом:

```
Var P : Array [1..12] Of Real;
```

Обработка массивов в программах производится покомпонентно.

Приведем примеры ввода значений в массивы:

```
For  I := 1 To 12 Do
     ReadLn(T[I]);
For  I := 1 To IMax Do
For  J := 1 To JMax Do
     ReadLn(Mas[I, J]);
```

Здесь каждое следующее значение будет вводиться с новой строки. Для построчного ввода используется оператор Read.

Аналогично в цикле по индексной переменной организуется вывод значений массива. Например:

```
For I := 1 To 12 Do Write(T[I]:8:4);
```

Следующий фрагмент программы организует построчный вывод матрицы на экран:

```
For I := 1 To IMax Do
Begin
  For J := 1 To JMax Do
      Write(Mas[I,J]:6);
  WriteLn
End;
```

После печати очередной строки матрицы оператор WriteLn без параметров переведет курсор в начало новой строки. Следует заметить, что в последнем примере матрица на экране будет получена в естественной форме прямоугольной таблицы, если JMax не превышает 12 (самостоятельно объясните, почему).

Рассмотрим несколько примеров типовых программ обработки массивов.

Пример 2.18. Для заданного ранее массива среднемесячных температур *T*[1 ... 12] требуется вычислить среднегодовую температуру и ежемесячные отклонения от этого значения.

Программа решения данной задачи следующая:

```
Program Example;
Cont N = 12;
```

```
Type Vec = Array [1..N] Of Real;
Var  T, Dt : Vec;
     St : Real;
     I : Integer;
Begin {---- Ввод исходных данных----}
     WriteLn('Вводите таблицу температур');
     For I := 1 To N Do
     Begin
          Write(I : 2, ':');
          ReadLn(T[I])
     End;
{----Вычисление средней температуры----}
     St := 0;
     For I := 1 To N Do
          St := St + T[I];
          St := St/N;
{----Вычисление таблицы отклонений от среднего----}
     For I := 1 To N Do
          Dt[I] := St-T[I];
{----Вывод результатов----}
     WriteLn('Средняя температура равна ', St:6:2);
     WriteLn;
     WriteLn('Отклонения от средней температуры:');
     For I := 1 To N Do
     WriteLn(I:1, ': ', Dt[I]:6:2)
End.
```

По этой программе можно рассчитать среднее значение и вектор отклонений от среднего значения для любого одномерного вещественного массива. Настройка размера массива осуществляется только редактированием раздела констант.

Пример 2.19. Требуется выбрать максимальный элемент из рассмотренного ранее массива температур, т. е отобрать самую высокую температуру и номер месяца, соответствующий ей.

Идея алгоритма решения этой задачи следующая: чтобы получить максимальную температуру в вещественной переменной TMax, сначала в нее заносится первое значение массива *T*[1]. Затем поочередно значение TMax сравнивается с остальными элементами массива температур, и каждое значение, большее TMax, присваивается этой переменной. Для получения номера самого теплого месяца в целую переменную NumMax следует засылать номер элемента массива температур каждый раз одновременно с занесением в TMax его значения.

Программа решения данной задачи следующая:

```
TMax := T[1];
NumMax := 1;
For I := 2 To 12 Do
```

```
  If T[I] > Tmax
  Then
  Begin
       TMax := T[I];
       NumMax := I
End;
```

Заметим, что если в массиве температур несколько значений, равных максимальному, то в NumMax будет получен первый номер из этих элементов. Для получения последней даты в операторе If следует заменить знак отношения «>» на «>=».

Пример 2.20. Требуется выполнить сортировку массива, т.е. в одномерном массиве X из N элементов требуется произвести перестановку значений таким образом, чтобы они расположились по возрастанию: $X_1 \le \le X_2 \le ... \le X_N$.

Существует целый класс алгоритмов сортировки. Опишем алгоритм, называемый методом пузырька.

Производится последовательное упорядочивание смежных пар элементов массива: X_1 и X_2, X_2 и X_3, ..., X_{N-1} и X_N. В итоге максимальное значение переместится в X_N. Затем ту же процедуру повторяют до получения X_{N-1} и т.д., вплоть до получения цепочки из двух элементов X_1 и X_2. Такой алгоритм имеет структуру двух вложенных циклов, причем внутренний цикл будет переменной (сокращающейся) длины.

Программа решения данной задачи следующая:

```
For I := 1 To N-1 Do
For K := 1 To N-I Do
  If X [K] > X [K + 1]
  Then
  Begin
       A := X[K];
       X[K] := X[K + 1];
       X[K + 1] := A
  End;
```

Пример 2.21. Для описанного ранее двухмерного массива среднемесячных температур за 10 лет требуется определить, в каком году было самое теплое лето, т.е. в каком году была наибольшая средняя температура летних месяцев.

Идея алгоритма решения следующая: сначала в векторе S получить средние температуры летних месяцев за 10 лет, а затем найти номер наибольшего элемента в этом векторе. Это и будет искомый год.

Программа решения данной задачи:

```
Program Example_2;
Type Month = Array[1..12] Of Real;
     Year = Array[1981..1990] Of Month;
```

```
Var  H: Year;
     S: Array[1981..1990] Of Real;
     I, J, K: Integer;
Begin {----Ввод данных с клавиатуры----}
  For I := 1981 To 1990 Do
  For J := 1 To 12 Do
  Begin
       Write(J : 2, '.', I : 4,': ');
       ReadLn(H[I, J])
  End;
{---Вычисление вектора средних летних температур---}
  For I := 1981 To 1990 Do
  Begin
       S[I] := 0;
       For J:= 6 To 8 Do
            S[I] := S[I] + H[I, J];
            S[I] := S[I]/3
  End;
{----Определение года с самым теплым летом----}
  K := 1981;
  For I := 1982 To 1990 Do
       If S[I] > S[K] Then K := I;
  WriteLn('Самое теплое лето было в ', K,'-м году')
End.
```

УПРАЖНЕНИЯ

1. Дан вектор $\{z_i\}$, $i = 1, ..., 50$. Вычислить его длину:

$$L = \sqrt{z_1^2 + z_2^2 + ... + z_{50}^2}.$$

2. Вычислить полином 10-й степени по формуле Горнера:

$$a_{10}x^{10} + a_9x^9 + ... + a_1x + a_0 = ((...(a_{10}x + a_9)x + a_8)x + ... + a_1)x + a_0.$$

3. Для вектора $\{x_i\}$, $i = 1, ..., 20$ найти количество компонентов, значения которых лежат в интервале $[0; 1]$.

4. Два заданных вектора $\{x_i\}$, $\{y_i\}$, $i = 1, ..., 10$, упорядоченных по возрастанию, слить в один вектор $\{z_i\}$, $i = 1, ..., 20$ таким образом, чтобы сохранилась их упорядоченность.

5. Для заданного массива, состоящего из 100 целых чисел, сначала вывести все числа, встречающиеся в нем несколько раз, а затем все числа, встречающиеся в нем только один раз.

6. Найти значение и индексы максимального элемента целочисленной матрицы размером 10 × 10.

7. Определить номер строки с наибольшим количеством нулей двоичной матрицы размером 5 × 10.

8. Все строки вещественной матрицы размером 10 × 15 упорядочить по убыванию значений их элементов.

9. Транспонировать целочисленную матрицу размером 5 × 5, т.е. отразить ее относительно главной диагонали.

10. В двоичной матрице размером 10 × 10 найти совпадающие строки.

2.17. РЕКУРСИВНЫЕ ПОДПРОГРАММЫ

Рекурсия — это способ организации вспомогательного алгоритма — подпрограммы, при котором эта подпрограмма (процедура или функция) в ходе выполнения обращается сама к себе.

Рекурсивным называется любой объект, который частично определяется через самого себя.

Например, рекурсивным является следующее определение двоичного кода:

```
<двоичный код> ::= <двоичная цифра> | <двоичный
код> <двоичная цифра>
<двоичная цифра> ::= 0 | 1
```

В рекурсивном определении обязательно должно присутствовать ограничение, т.е. *граничное условие*, при выходе на которое дальнейшая инициация рекурсивных обращений прекращается.

Приведем примеры рекурсивных определений некоторых математических функций.

Классическим примером рекурсии является определение факториала. С одной стороны, факториал определяется в виде: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, а с другой стороны, его можно определить следующим образом:

$$n! = \begin{cases} 1, & \text{если } n \le 1; \\ (n-1)!\,n, & \text{если } n > 1. \end{cases}$$

Граничным в данном случае является условие $n \le 1$.

Рекурсивное определение функции $K(n)$, возвращающей количество цифр в заданном натуральном числе n, имеет следующий вид

$$K(n) = \begin{cases} 1, & \text{если } n < 10; \\ K(n/10) + 1, & \text{если } n \ge 10. \end{cases}$$

Рекурсивное определение функции $S(n)$, вычисляющей сумму цифр заданного натурального числа, запишите самостоятельно по аналогии.

Рекурсивное определение функции $C(m, n)$, где $0 \le m \le n$, для вычисления биномиального коэффициента C_n^m имеет следующий вид:

$$C_n^0 = C_n^n = 1; \quad C_n^m = C_{n-1}^m + C_{n-1}^{m-1}.$$

Обращение к рекурсивной подпрограмме ничем не отличается от вызова любой другой подпрограммы. При этом при каждом новом рекурсивном обращении в памяти создается новая копия подпрограммы со всеми локальными переменными. Такие копии будут создаваться до момента выхода на граничное условие. Очевидно, что в случае отсутствия граничного условия, неограниченный рост числа таких копий приведет к аварийному завершению программы, т.е. переполнению стека.

Порождение все новых копий рекурсивной подпрограммы до выхода на граничное условие называется *рекурсивным спуском*. Максимальное число копий рекурсивной подпрограммы, одновременно находящихся в памяти компьютера, называется *глубиной рекурсии*. Завершение работы рекурсивных подпрограмм, вплоть до самой первой, инициировавшей рекурсивные вызовы, называется *рекурсивным подъемом*.

Выполнение действий в рекурсивной подпрограмме может быть организовано одним из следующих способов:

Рекурсивный подъем	Рекурсивный спуск	Рекурсивный спуск и рекурсивный подъем

Begin
  P;
  операторы;
End;
	Begin
	  операторы;
	  P
	End;
		Begin
		  операторы;
		  P;
		  операторы
		End;

Здесь P — рекурсивная подпрограмма.

Действия в рекурсивной подпрограмме могут выполняться либо на одном из этапов рекурсивного обращения, либо на обоих сразу. Способ организации действий диктуется логикой разрабатываемого алгоритма.

Реализуем приведенные ранее рекурсивные определения в виде функций и процедур на Паскале.

Пример 2.22. Программные реализации рекурсивного определения факториала:

```
{Функция на Паскале}
Function Factrial (N : Integer) : Extended;
Begin
  If N <= 1
  Then Factorial := 1
  Else Factrial := Factorial(N-1) * N
End;

{Процедура на Паскале}
Procedure Factorial (N : Integer; Var F : Extended);
Begin
  If N <= 1
  Then F := 1
  Else Begin
         Factorial(N-1, F);
         F := F * N
       End
End;
```

Пример 2.23. Программные реализации рекурсивного определения функции $K(N)$, возвращающей количество цифр в заданном натуральном числе N:

```
{Функция на Паскале}
Function K(N : Longint) : Byte;
Begin
  If N < 10
  Then K := 1
  Else K := K(N Div 10) + 1
End;

{Процедура на Паскале}
Procedure K(N : Longint; Var Kol : Byte)
Begin
  If N < 10
  Then Kol := 1
  Else Begin
         K(N Div 10, Kol);
         Kol := Kol + 1
       End
End;
```

Пример 2.24. Программные реализации рекурсивного вычисления биномиальных коэффициентов:

```
{Функция на Паскале}
Function C(m, n : Byte) : Longint;
Begin
  If (m = 0) Or (m = n)
  Then C := 1
  Else C := C(m, n-1) + C(m-1, n-1)
End;

{Процедура на Паскале}
Procedure C(m, n : Byte; Var R : Longint);
Var R1, R2: Longint;
Begin
  If (m = 0) Or (m = n)
  Then R := 1
  Else Begin
         C(m,n-1, R1);
         C(m-1, n-1, R2);
         R := R1 + R2
       End
End;
```

Пример 2.25. Вычисление суммы элементов линейного массива.

При решении данной задачи используем следующее утверждение: сумма равна нулю, если число элементов равно нулю, и равна сумме всех предыдущих элементов плюс последний, если число элементов не равно нулю.

Программа будет иметь следующий вид:

```
{Программа на Паскале }
Program Rec2;
Type LinMas = Array[1..100] Of Integer;
Var A : LinMas;
    I, N : Byte;

{Рекурсивная функция}
Function Summa(N : Byte; A : LinMas): Integer;
Begin
    If N = 0
    Then Summa := 0
    Else Summa := A[N] + Summa(N-1, A)
End;

{Основная программа}
Begin
    Write('Количество элементов массива? ');
    ReadLn(N);
    Randomize;
    For I := 1 To N Do
    Begin
```

```
        A[I] := -10 + Random(21);
        Write(A[I] : 4)
    End;
    WriteLn;
    WriteLn('Сумма: ', Summa(N, A))
End.
```

Пример 2.26. Требуется определить, является ли заданная строка палиндромом, т.е. читается одинаково слева направо и справа налево. Граничное условие: строка является палиндромом, если она пустая или состоит из одного символа.

Идея решения данной задачи заключается в просмотре строки одновременно слева направо и справа налево и в сравнении соответствующих символов. Если в какой-то момент символы не совпадают, делается вывод о том, что строка не является палиндромом, если же удается достичь середины строки и при этом все соответствующие символы совпали, то строка является палиндромом.

```
Program Palindrom;
{Рекурсивная функция}
Function Pal(S : String): Boolean;
Begin
  If Length(S) <= 1
  Then Pal := True
  Else Pal := (S[1] = S[Length(S)]) And
       Pal(Copy(S, 2, Length(S) - 2));
End;
Var S : String;

{Основная программа}
Begin
  Write('Введите строку: ');
  ReadLn(S);
  If Pal(S)
  Then WriteLn('Строка является палиндромом')
  Else WriteLn('Строка не является палиндромом')
End.
```

Теперь, используя аналогичный подход, самостоятельно составьте программу определения, является ли заданное натуральное число палиндромом.

Подводя итог, заметим, что использование рекурсии является красивым приемом программирования. В то же время при решении большинства практических задач этот прием неэффективен, так как увеличивает время исполнения программы и зачастую требует значительного объема памяти для хранения копий подпрограммы на рекурсивном спуске. Следовательно, на практике разумно заменять рекурсивные алгоритмы на итеративные.

УПРАЖНЕНИЯ

1. Привести собственные примеры содержательных задач, где для решения может быть использован рекурсивный вспомогательный алгоритм.

2. Определить, почему следующий алгоритм посимвольного формирования строки завершится аварийно:

```
Function Stroka : String;
Var C : Char;
Begin
  Write('Введите очередной символ: '); ReadLn(C);
  Stroka := Stroka + C
End;
```

На каком этапе выполняются действия в этом алгоритме?

3. Описать рекурсивную функцию pow(x, n) от вещественного x ($x \neq 0$) и целого n, вычисляющую величину x^n по следующей формуле:

$$x^n = \begin{cases} 1 & \text{при } n = 0; \\ \dfrac{1}{x^{-n}} & \text{при } n < 0; \\ x \cdot x^{n-1} & \text{при } n > 0. \end{cases}$$

4. Даны натуральные числа n и m. Найти НОД(n, m), используя программу, включающую в себя рекурсивную процедуру вычисления НОД, основанную на соотношении НОД(n, m) = НОД(m, r), где r — остаток от деления n на m.

2.18. МНОЖЕСТВА

Одним из фундаментальных разделов математики является теория множеств. Некоторые моменты математического аппарата этой теории реализованы в Паскале через *множественный тип данных* (множества).

Множеством называется совокупность однотипных элементов, рассматриваемых как единое целое. В Паскале могут быть только конечные множества. В Турбо Паскале множество может содержать от 0 до 255 элементов.

В отличие от массива элементы множества не пронумерованы и не упорядочены. Каждый отдельный элемент множества не идентифицируется и с ним нельзя выполнять какие-либо действия. Действия могут выполняться только над множеством в целом.

Тип элементов множества называется базовым типом. Базовым может быть любой скалярный тип за исключением типа Real.

Конструктор множества. Конкретные значения множества задаются с помощью конструктора множества, представляющего собой список элементов, заключенный в квадратные скобки. Сами элементы могут быть либо константами, либо выражениями базового типа.

Приведем несколько примеров задания множеств с помощью конструктора:

```
[3, 4, 7, 9, 12]  — множество из пяти целых чисел;
       [1..100]   — множество целых чисел от 1 до 100;
   ['a','b','c']  — множество, содержащее три литеры
                    a, b, c;
['a'..'z','?','!'] — множество, содержащее все строчные
                    латинские буквы, а также знаки ? и !.
```

Символом «[]» обозначают пустое множество, т.е. множество, не содержащее никаких элементов.

Не имеет значения порядок записи элементов множества внутри конструктора. Например, [1, 2, 3] и [3, 2, 1] — это эквивалентные множества.

Каждый элемент в множестве учитывается только один раз, поэтому множества [1, 2, 3, 4, 2, 3, 4, 5] и [1..5] эквивалентны.

Переменные множественного типа описываются следующим образом:

```
Var <идентификатор> : Set Of <базовый тип>.
```

Например:

```
Var A, D : Set Of Byte;
    B : Set Of 'a'..'z';
    C : Set Of Boolean;
```

Нельзя вводить значения в множественную переменную оператором ввода и выводить оператором вывода. Множественная переменная может получить конкретное значение только в результате выполнения оператора присваивания следующего формата:

```
<множественная переменная> := <множественное выражение>
```

Например:

```
A := [50,100,150,200];
B := ['m','n','k'];
```

```
C := [True,False];
D := A;
```

Кроме того, выражения могут включать в себя операции над множествами.

Операции над множествами. В Паскале реализованы основные операции математической теории множеств: объединение, пересечение, разность. Во всех этих операциях операнды и результаты есть множественные величины одинакового базового типа.

Объединением двух множеств A и B называется множество, состоящее из элементов, входящих хотя бы в одно из множеств *A* или *B*. Знак операции объединения в Паскале — это «+».

Результат объединения двух множеств схематически показан на рис. 2.27, *а*. Например:

```
[1,2,3,4] + [3,4,5,6] → [1,2,3,4,5,6].
```

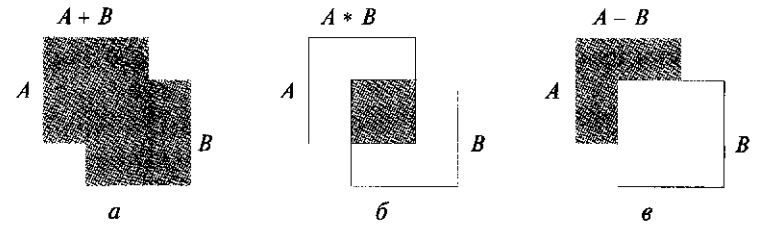
Пересечением двух множеств A и B называется множество, состоящее из элементов, одновременно входящих и в множество *A*, и в множество *B* (рис. 2.27, *б*). Например:

```
[1,2,3,4] * [3,4,5,6] → [3,4].
```

Разностью двух множеств A и B называется множество, состоящее из элементов множества *A*, не входящих в множество *B* (рис. 2.27, *в*). Например:

```
[1,2,3,4] - [3,4,5,6] → [1,2]
[3,4,5,6] - [1,2,3,4] → [5,6]
```

Очевидно, что операции объединения и пересечения переста-новочные, а разность множеств — операция неперестановочная.



а *б* *в*

Рис. 2.27. Операции над множествами:
а — объединение; *б* — пересечение; *в* — разность

Таблица 2.9. Операции отношения над множествами

Отношение	Результат	
	True	False
$A = B$	Множества A и B совпадают	В противном случае
$A <> B$	Множества A и B не совпадают	То же
$A <= B$	Все элементы A принадлежат B	»
$A >= B$	Все элементы B принадлежат A	»

Множества можно сравнивать между собой, т.е. для них определены *операции отношения*. Результатом отношения, как известно, является логическая величина True или False. Для множеств применимы все операции отношения, за исключением «>» и «<».

Операции отношения над множествами описаны в табл. 2.9. При этом предполагается, что множества A и B содержат элементы одного типа.

Приведем несколько примеров выполнения операций отношения. Пусть переменная M описана в программе следующим образом:

```
Var M : Set Of Byte;
```

В разделе операторов ей присваивается значение

```
M := [3,4,7,9];
```

Тогда операции отношения дадут следующие результаты:

```
M = [4, 7, 3, 3, 9,] -     True;
  M <> [7, 4, 3, 9, ] -    False;
          [3, 4] <= M -    True;
              [] <= M -    True;
        M >= [1..10] -     False;
          M <= [3..9] -    True.
```

Операция *вхождения* устанавливает связь между множеством и скалярной величиной, тип которой совпадает с базовым типом множества, т.е. если x — скалярная величина данного типа, а M — множество, то операция вхождения записывается в виде

```
x In M.
```

Результатом здесь будет логическая величина True, если значение *x* входит в множество *M*, и False — в противном случае. Для описанного ранее множества

```
4 In M — True,
5 In M — False.
```

Рассмотрим несколько задач, для решения которых удобно использовать множества.

Пример 2.27. Дана символьная строка. Требуется определить в ней количество знаков препинания (. — , * : ! ?).

Программа для решения данной задачи:

```
Program P1;
Var S: String; I, K: Byte;
Begin
  ReadLn(S); K := 0;
  For I := 1 To Length(S) Do
  If S[I] In ['.', '-', ',', ';', ':', '!', '?']
  Then K := K + 1;
  WriteLn('Число знаков препинания равно ', K)
End.
```

В этом примере использована множественная константа с символьным типом элементов. Однако задачу можно решить и без множества, записав в операторе If длинное логическое выражение `(S[I] = '.') Or (S[I] = '—')` и т.д. Использование множества сокращает запись.

Пример 2.28. Даны две символьные строки, содержащие только строчные латинские буквы. Требуется построить строку *S*3, в которую войдут только общие символы *S*1 и *S*2 в алфавитном порядке и без повторений.

Программа для решения данной задачи:

```
Program P2;
Type Mset = Set Of 'a'..'z';
Var S1, S2, S3 : String;
    MS1, MS2, MS3 : Mset;
    C : Char;
Procedure SM(S : String; Var MS : Mset);
{Процедура формирует множество MS, содержащее все символы строки S}
Var I: Byte;
Begin MS := [];
  For I := 1 To Length(S) Do
      MS := MS + [S[I]]
End;
```

```
Begin {Ввод исходных строк}
  ReadLn(S1); ReadLn(S2);
{Формирование множеств MS1 и MS2 из символов строк S1 и
S2}
  SM(S1, MS1); SM(S2, MS2);
{Пересечение множеств - выделение общих элементов в мно-
жество MS3}
  MS3 := MS1 * MS2;
{Формирование результирующей строки S3}
  S3 := '';
  For C := 'a' To 'z' Do
      If C In MS3 Then S3 := S3 + C;
  WriteLn('Результат: ', S3)
End.
```

Пример 2.29. Требуется составить программу, по которой из последовательности натуральных чисел от 2 до N ($1 < N \le 255$) будут выбраны все простые числа.

Существует алгоритм, известный под названием «Решето Эратосфена», суть которого состоит в следующем:

1) из числовой последовательности выбирают минимальное значение — простое число;

2) из последовательности удаляют все числа, кратные выбранному значению;

3) если после удаления всех кратных чисел последовательность не стала пустой, возвращаются к выполнению первого пункта.

Приведем пример работы такого алгоритма для $N = 15$ (здесь подчеркнуты выбранные простые числа):

2 3 4 5 6 7 8 9 10 11 12 13 14 15
3 5 7 9 11 13 15
5 7 11 13
7 11 13
11 13
13

В программе решения данной задачи удобно использовать множественный тип данных:

```
Program Eratosfen;
Const N = 201;
{Возможно любое значение 1 < N < 256}
Var A, B : Set Of 2..N;
K, P : Integer;
Begin
{Формирование исходного множества A; B — искомое множе-
ство простых чисел, сначала — пустое}
    A := [2..N];
    B := [];
    P := 2;
```

```
    Repeat
{Поиск минимального числа в множестве A}
          While Not (P In A) Do P := P + 1;
{Включение найденного числа в множество B}
          B := B + [P];
          K := P;
{Исключение из A чисел, кратных P}
          While K <= N Do
          Begin
                A := A-[K];
                K := K + P;
          End
    Until A = [];
{Вывод результата, т.е. всех чисел из множества B в по-
рядке возрастания}
    For P := 2 To N Do If P In B Then WriteLn(P)
End.
```

Это красивая программа. Но, к сожалению, ее нельзя использовать при $N > 255$ из-за отмеченного выше ограничения в Турбо Паскале на максимальный размер множества.

Как уже говорилось, нельзя вводить значения непосредственно в множество. Однако такая потребность у программиста может возникнуть. В этом случае можно использовать процедуру INSET.

Для примера рассмотрим множество с символьным базовым типом, предполагая, что в основной программе глобально объявлен тип SetChar:

```
Type SetChar : Set Of Char;
Procedure INSET(Var M : SetChar);
Var I, N : Byte; C : Char;
Begin
  Write('Укажите размер множества: '); ReadLn(N);
  M := [];
  For I := 1 To N Do
  Begin
       Write(I:1,'-й элемент: '); ReadLn(C);
       M := M + [C]
  End;
  WriteLn('Ввод завершен!')
End;
```

В основной программе для ввода значений в множество, например с именем SIM, достаточно записать оператор INSET(SIM). В результате произойдет диалоговый ввод значений.

УПРАЖНЕНИЯ

1. В программе присутствуют следующие описания:

```
Var P: Set Of 0..9; I, J : Integer;
```

Определить, какое значение получит переменная *P* при выполнении следующих операторов присваивания, если $i = 3$ и $j = 5$:

а) `P := [I + 3, J Div 2, J..Sqr(I) - 3];`
б) `P := [2 * I..J];`
в) `P := [I, J, 2 * I, 2 * J]?`

2. Вычислить значения следующих отношений:

а) `[2]<>[2,2,2];`
б) `['a','b'] = ['b','a'];`
в) `[4,5,6] = [4,5,6];`
г) `['c','b'] = ['c'..'b'];`
д) `[2,3,5,7] <= [1..9];`
е) `[3,6..8] <= [2..7,9];`
ж) `Trunc(3.9) In[1,3,5];`
з) `Odd(4) In [].`

3. Вычислить значения следующих выражений:

а) `[1,3,5] + [2,4];`
б) `[1,3,5] * [2,4];`
в) `[1,3,5] - [2,4];`
г) `[1..6] + [3..8];`
д) `[1..6] * [3..8];`
е) `[1..6] - [3..8];`
ж) `[] + [4];`
з) `[] * [4];`
и) `[] - [4].`

4. Составить программу подсчета количества различных значащих цифр в десятичной записи натурального числа.

5. Составить программу, печатающую в возрастающем порядке все целые числа из интервала 1 ... 255, представленные в виде $n^2 + m^2$, где n, $m > 0$.

6. Дана строка из строчных русских букв. Между соседними словами — запятая, за последним словом — точка. Составить программу, печатающую в алфавитном порядке:

а) все гласные буквы, которые входят в каждое слово;
б) все согласные буквы, которые не входят ни в одно слово;
в) все согласные буквы, которые входят только в одно слово;
г) все гласные буквы, которые входят более чем в одно слово.

2.19. ФАЙЛЫ

С термином «файл» вам уже приходилось встречаться. Прежде всего это понятие обычно связывают с информацией на устройствах внешней памяти.

В Паскале понятие файла имеет два значения:

- поименованная информация на внешнем устройстве (внешний файл);
- переменная файлового типа в Паскаль-программе (внутренний файл).

В программе между этими объектами устанавливается связь. Вследствие этого все, что происходит в процессе выполнения программы с внутренним файлом, дублируется во внешнем файле. С элементами файла можно выполнять только две операции: считывать из файла, записывать в файл.

Файловый тип переменной — структурированный тип, представляющий собой совокупность однотипных элементов, число которых заранее (до исполнения программы) не определено.

Структура описания файловой переменной имеет вид

```
Var <имя переменной> : File Of <тип элемента>;
```

Здесь <тип элемента> может быть любым, кроме файлового. Например:

```
Var  Fi : File Of Integer;
     Fr : File Of Real;
     Fc : File Of Char;
```

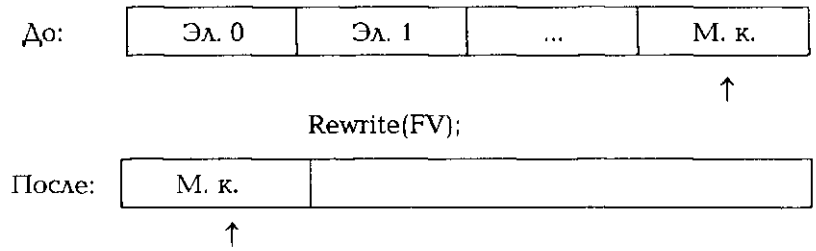
Файл можно представить как последовательную цепочку элементов (Эл.), пронумерованных от 0, заканчивающуюся специальным кодом, называемым *маркером конца* (М.к.):

Эл. 0	Эл. 1	...	Эл. *N*	М. к.

Число элементов, хранящихся в данный момент в файле, называется его *текущей длиной*. Существует специальная ячейка памяти, которая хранит адрес элемента файла, предназначенного для текущей обработки (записи или считывания). Этот адрес называется *указателем*, или *окном*, файла.

Для того чтобы начать запись в файл, его следует *открыть для записи*. Это обеспечивает процедура Rewrite(FV); (где FV — имя файловой переменной). При этом указатель устанавливается на начало файла. Если в файле до этого была информация, она исчезает.

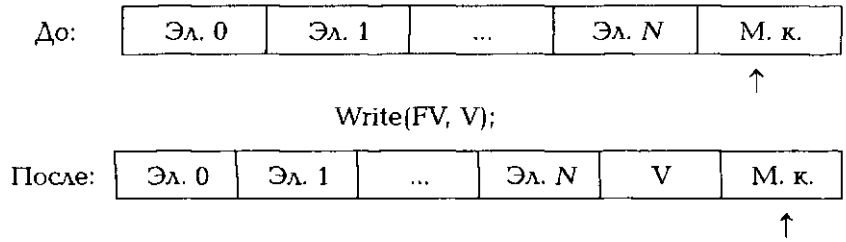
Схематически выполнение процедуры Rewrite можно представить в следующем виде:



Стрелкой снизу отмечается позиция указателя.

Запись в файл осуществляется процедурой Write(FV, V); (где V — переменная того же типа, что и файл F V). Запись происходит в том месте, на которое установлено окно (указатель файла). Сначала записывается значение, затем указатель смещается в следующую позицию. Если новый элемент вносится в конец файла, сдвигается маркер конца.

Схема выполнения данной процедуры следующая:



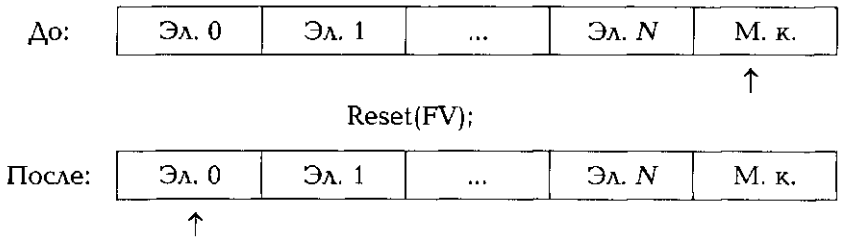
Пример 2.30. Требуется в файловую переменную Fx внести 20 вещественных чисел, последовательно вводимых с клавиатуры.

Программа решения следующая:

```
Var Fx : File Of Real;
    X : Real; I: Byte;
Begin
  Rewrite(Fx);
  For I := 1 To 20 Do
  Begin
       Write('? '); ReadLn(X);
       Write(Fx, X)
  End
End.
```

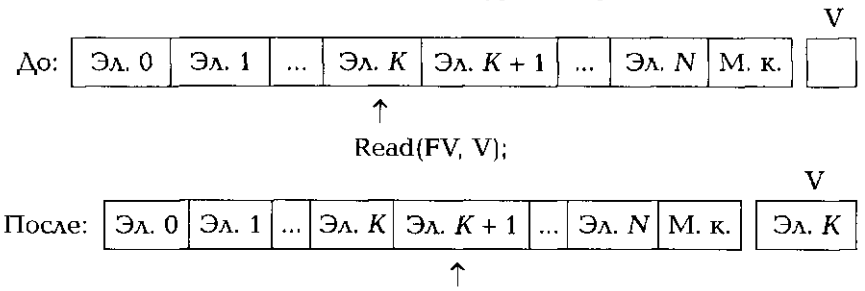
Для считывания элементов файла с его начала следует *открыть файл для считывания*, что выполняется процедурой Reset(FV). В результате указатель устанавливается на начало файла. При этом вся информация в файле сохраняется.

Схема выполнения данной процедуры следующая:



Считывние из файла осуществляется процедурой Read(FV, V); (где V — переменная того же типа, что и файл FV). При этом значение текущего элемента файла записывается в переменную V, а указатель смещается к следующему элементу.

Схема выполнения данной процедуры следующая:



Доступ к элементам файла может быть *последовательным* или *прямым*. В стандартном Паскале допускается только последовательный доступ к элементам.

Принцип последовательного доступа: для того чтобы прочитать *n*-ю запись файла, сначала следует прочитать все предыдущие записи с 1-й по $(n - 1)$-ю.

Пример 2.31. Требуется в переменной *X* получить 10-й элемент вещественного файла *Fx*.

Программа решения следующая:

```
Program A;
Var Fx : File Of Real;
    X : Real;
Begin
  Reset(Fx);
  For I := 1 To 10 Do Read(Fx, X)
End.
```

Функция Eof(FV) проверяет маркер конца файла (end of file). Это логическая функция, которая получает значение True, если

указатель установлен на маркер конца, и значение False — в противном случае.

Пример 2.32. Требуется просуммировать все числа из файла Fx, описанного в примере 2.31.

Программа решения следующая:

```
Reset(Fx);
Sx := 0;
While Not Eof(Fx) Do
Begin
  Read(Fx, X);
  Sx := Sx + X
End;
```

Также решить данную задачу можно с помощью цикла Repeat:

```
Repeat
   Read(Fx, X);
   Sx := Sx + X
Until Eof(Fx);
```

Во втором варианте возможна ошибка считывания, если файл Fx пустой. Первый вариант от такой ошибки застрахован, поэтому он предпочтительнее.

Внешние файлы. В Турбо Паскале все внешние устройства (дисплей, клавиатура, принтер, диски и др.) трактуются как логические устройства с файловой структурой организации данных. Все немагнитные внешние устройства — однофайловые. Иначе говоря, с каждым из них связан один файл со стандартным именем, предназначенный для обмена с внутренней памятью ЭВМ текстовой (символьной) информацией.

Стандартные имена логических устройств определяются операционной системой, в среде которой работает Паскаль. В системе MS DOS определены следующие имена:

CON (консоль) — логическое устройство, связанное при вводе с клавиатурой, при выводе — с экраном;

PRN (принтер) — логическое имя файла, связанного с устройством печати;

AUX — логическое имя коммуникационного канала, который используется для связи ПК с другими машинами;

INPUT — стандартное устройство ввода, связанное с клавиатурой, причем вводимые с клавиатуры символы отражаются на экране дисплея;

OUTPUT — стандартное устройство вывода на экран.

Магнитный диск (МД) — многофайловое устройство. На нем могут храниться как стандартные (системные) файлы, так и файлы, создаваемые пользователем. На магнитном диске могут создаваться файлы любых типов. Файлы на МД могут использоваться как в режиме считывания, так и в режиме записи.

Список файлов на диске хранится в директории (каталоге) диска. Каталог вызывается на экран системной командой DIR. В полной форме каталог содержит идентификаторы файлов, объем занимаемой памяти, дату и время создания файла. Идентификатор файла состоит из имени и типа файла:

```
<имя файла>.<тип файла>
```

Имя файла содержит от одной до восьми латинских букв и (или) цифр; тип файла — необязательный элемент (от нуля до трех символов), указывающий на характер информации, хранимой в файле. Например:

PROGRAM.PAS — в файле текст программы на Паскале;

NUMBER.DAT — файл числовых данных;

NAMES.TXT — текстовый файл.

Для организации связи между файловой переменной и внешним файлом в Турбо Паскале используется следующая *процедура назначения*:

```
Assign(<имя файловой переменной>,
    <идентификатор внешнего файла>).
```

Здесь <идентификатор внешнего файла> — строковая величина (константа или переменная). Например:

```
Assign(Fi, 'Number.dat');
```

После выполнения процедур Assign и Rewrite создается новый внешний файл, имя которого заносится в директорию.

Если файл открывается для считывания (Assign и Reset), то в указанном каталоге уже должен содержаться указанный внешний файл. В противном случае будет обнаружена ошибка.

Работа с файлом в программе завершается его *закрытием* с помощью следующей процедуры:

```
Close(<имя файловой переменной>);
```

Например:

```
Close(Fi).
```

Итак, последовательность действий при создании и заполнении файла:

- описать файловую переменную;
- описать переменную того же типа, что и файл;
- произвести назначение (Assign);
- открыть файл для записи (Rewrite);
- записать в файл данные (Write);
- закрыть файл (Close).

Пример 2.33. Требуется создать файл, содержащий среднесуточные температуры за некоторое количество дней. При этом не обязательно предварительно указывать количество чисел во вводимой информации. Можно договориться о каком-то условном значении, которое будет признаком конца ввода, например число 9 999.

Программа решения следующая:

```
Program Task1;
Var Ft : File Of Real; T : Real;
Begin
  Assign(Ft, 'Temp.dat'); Rewrite(Ft);
  WriteLn('Вводите данные. Признак конца - 9999');
  ReadLn(T);
  While T <> 9999 Do
  Begin
       Write(Ft, T); Write('? '); ReadLn(T)
  End;
  WriteLn('Ввод данных закончен!');
  Close(Ft)
End.
```

В результате работы этой программы на диске будет создан файл с именем Temp.dat, в котором сохранится введенная информация.

Для последовательного считывания данных из файла требуется выполнить также действия:

- описать файловую переменную;
- описать переменную того же типа;
- выполнить назначение (Assign);
- открыть файл для считывания (Reset);
- в цикле считывать из файла (Read);
- закрыть файл (Close).

Пример 2.34. Требуется определить среднюю температуру для значений, хранящихся в файле Temp.dat.

Программа решения следующая:

```
Program Task2;
Var  Ft : File Of Real;
     T, St : Real; N : Integer;
Begin Assign(Ft, 'Temp.dat');
  Reset(Ft);
  St := 0;
  While Not Eof(Ft) Do
  Begin
       Read(Ft, T);
       St := St + T
  End;
  N := FileSize(Ft);
  St := St/N;
  WriteLn('Средняя температура за ',
  N:3, ' суток равна', St:7:2,' градусов');
  Close(Ft)
End.
```

В этой программе использована следующая функция определения размера файла:

```
FileSize(<имя файловой переменной>);
```

Результат выполнения этой функции — целое число, равное текущей длине файла.

Примечание. Согласно стандартному языку Паскаль в файл, открытый оператором Rewrite, можно только записывать информацию, а файл, открытый оператором Reset, можно использовать только для считывания. В Турбо Паскале допускается запись (Write) в файл, открытый для считывания (Reset), что создает определенные удобства для модификации файлов.

Текстовые файлы. Это наиболее часто употребляемая разновидность файлов. Как уже отмечалось ранее, немагнитные внешние устройства (логические) работают только с текстовыми файлами. Файлы, содержащие тексты программ на Паскале и других языках программирования, являются текстовыми. Различная документация и информация, передаваемые по каналам электронной связи, — все это текстовые файлы.

В программе файловая переменная текстового типа описывается следующим образом:

```
Var <идентификатор> : text;
```

Текстовый файл представляет собой символьную последовательность, разделенную на строки. Каждая строка символов (S) заканчивается специальным кодом — маркером конца строки (М.к.с.). Весь файл заканчивается маркером конца файла (М.к.ф.). Схематически это можно представить следующим образом:

S_1	S_2	...	S_{k1}	М.к.с.	S_1	S_2	...	S_{k2}	М.к.с.	...	М.к.ф.

При этом каждый символ представлен во внутреннем коде (ASCII) и занимает 1 байт. Однако не только делением на строки отличается текстовый файл от символьного: *в текстовый файл можно записать, а также считать из него информацию любого типа*. Если эта информация символьная, то в процессе считывания или записи происходит ее преобразование из символьной формы во внутреннюю, и обратно.

Текстовый файл можно создать или преобразовать с помощью текстового редактора, его также можно просмотреть на экране дисплея или распечатать на принтере.

В программах на Паскале для работы с текстовыми файлами наряду с процедурами Read и Write употребляются процедуры ReadLn и WriteLn.

Процедура, считывающая строку из файла с именем FV и помечающая прочитанное в переменные из списка ввода:

```
ReadLn(FV, <список ввода>).
```

Процедура, записывающая в файл FV значения из списка вывода и выставляющая маркер конца строки:

```
WriteLn(FV, <список вывода>).
```

Функция Eoln(FV) используется для обнаружения конца строки в текстовом файле (end of line — конец строки). Это логическая функция, которая принимает значение True, если указатель файла достиг маркера конца строки, и значение False — в противном случае.

Употребление операторов Read и ReadLn без указания имени файловой переменной означает считывание из стандартного файла Input (ввод с клавиатуры). Употребление операторов Write и WriteLn без имени файловой переменной означает запись в стандартный файл Output (вывод на экран). Эти варианты операторов мы уже многократно использовали. Считается, что файлы Input и Output открываются соответственно для считывания и записи при работе любой программы.

При вводе с клавиатуры маркер конца строки обнаруживается при нажатии клавиши <Enter>.

Процедура ReadLn может использоваться без списка ввода. В этом случае происходит пропуск текущей строки в считываемом файле.

Употребление процедуры WriteLn без списка вывода означает вывод пустой строки (в файле выставляется маркер конца строки).

При записи в текстовый файл в списке вывода могут присутствовать форматы. Действия форматов уже рассматривались при описании вывода данных на экран (см. подразд. 2.6). Точно также форматы работают и при выводе в текстовые файлы, связанные с любыми другими устройствами.

Пример 2.35. Пусть файл с именем Note.txt содержит некоторый текст. Требуется определить количество строк в этом тексте.

Программа решения следующая:

```
Var Note : Text; K : Integer;
Begin
  Assign(Note, 'Note.txt');
  Reset(Note);
  K := 0;
  While Not Eof(Note) Do
  Begin
    ReadLn(Note);
    K := K + 1
  End;
  WriteLn('Количество строк равно ', K);
  Close(Note)
End.
```

Используемый здесь оператор ReadLn(Note) «пролистывает» строки из текстового файла Note, не занося их в какую-либо переменную.

Пример 2.36. Требуется в текстовом файле Note.txt определить длину самой большой строки.

Программа решения следующая:

```
Var Note : Text;
    Max, K : Integer; C: Char;
Begin
  Assign(Note, 'Note.txt');
  Reset(Note);
  Max := 0;
  While Not Eof(Note) Do
  Begin
    K := 0;
    While Not Eoln(Note) Do
    Begin
```

```
      Read(Note, C);
      K := K + 1
    End;
    If K > Max Then Max := K;
    ReadLn(Note)
  End;
  WriteLn('Наибольшая строка имеет ', Max, ' знаков');
  Close(Note)
End.
```

Здесь каждая строчка прочитывается посимвольно, при этом в переменной *K* работает счетчик числа символов в строке, а в переменной Max отбирается наибольшее значение счетчика.

Пример 2.37. Требуется решить следующую задачу. Под действием силы *F* с начальной скоростью *V* в вязкой среде движется тело массой *M*. Сопротивление движению пропорционально квадрату скорости с коэффициентом *K*. Определить время прохождения пяти контрольных точек траектории движения, расстояние до которых от точки старта заданы.

Пусть с помощью текстового редактора в файле DATE.TXT исходные данные сформированы в виде следующей таблицы:

Исходные данные				
M (кг)	*F* (*H*)	*V* (м/с)	*K* (кг/м)	
36,3	2000	50,5	0,5	
Координаты контрольных точек (м)				
X(1)	*X*(2)	*X*(3)	*X*(4)	*X*(5)
10	100	150	1 000	3 000

Следует ввести числовые данные в вещественные переменные *M*, *F*, *V*, *K* и массив *X*[1..5], произвести расчеты, получить результаты в массиве *T*[1..5], вывести их на экран, а также в текстовый файл на диске с именем Result.txt.

Приведем программу решения без расчетной части, показав только ввод исходных данных и вывод результатов:

```
Var M, F, V, K : Real; I : Integer;
    T, X : Array[1..5] Of Real;
    FR, FD : Text;
    Begin
       Assign(FD, 'DATE.TXT'); Reset(FD);
       Assign(FR, 'Result.txt'); Rewrite(FR);
{----Пропуск первых трех строк----}
       ReadLn(FD); ReadLn(FD); ReadLn(FD);
{----Ввод данных----}
```

```
        ReadLn(FD, M, F, V, K);
{----Пропуск трех строк----}
        ReadLn(FD); ReadLn(FD); ReadLn(FD);
{----Ввод данных----}
        For I := 1 To 5 Do Read(FD, X[I]);
        ..............
        {РАСЧЕТНАЯ ЧАСТЬ ПРОГРАММЫ}
        ..............
{----Вывод результатов на экран и в файл FR----}
        WriteLn('Результаты '); WriteLn;
        WriteLn(FR, ' Результаты'); WriteLn(FR);
        WriteLn(' T(1) T(2) T(3) T(4) T(5)');
        WriteLn(FR, ' T(1) T(2) T(3) T(4) T(5)');
        For I := 1 To 5 Do
        Begin
                Write(T[I]:8:2); Write(FR, T[I]:8:2)
        End;
        Close(FD); Close(FR)
    End.
```

Результаты будут сохранены в файле Result.txt. Их можно посмотреть на экране, распечатать на бумаге. При необходимости этот файл может стать входным для другой программы, если содержащаяся в нем информация будет являться исходной для решения другой задачи.

УПРАЖНЕНИЯ

1. Дан файл вещественных чисел. Определить количество нулевых значений в этом файле.
2. Даны два файла целых чисел. Определить, являются ли они тождественными.
3. Даны два символьных файла одинакового размера. Произвести обмен информацией между ними.
4. Имеется внешний текстовый файл. Напечатать первую из самых коротких его строк.
5. Описать процедуру Lines(T), которая построчно печатает содержимое непустого текстового файла T, вставляя в начало каждой печатаемой строки ее порядковый номер, занимающий четыре позиции, и пробел.
6. В текстовом файле T записана непустая последовательность вещественных чисел, разделенных пробелами. Описать функцию Max(T) для нахождения наибольшего из этих чисел.

2.20. КОМБИНИРОВАННЫЙ ТИП ДАННЫХ

Все структурированные типы данных, которые уже рассматривались, представляют собой совокупности однотипных величин.

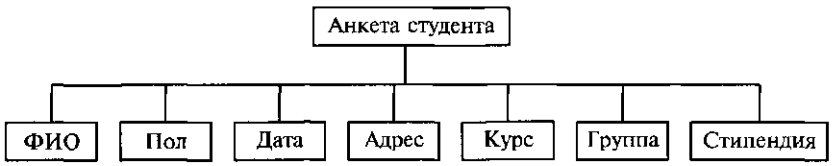


Рис. 2.28. Одноуровневое дерево

Комбинированный тип данных — структурированный тип, состоящий из фиксированного числа компонентов (полей) разного типа. Другое его название — *запись*.

Обычно запись содержит совокупность разнотипных атрибутов, относящихся к одному объекту. Например, анкетные сведения о студенте вуза можно представить в виде информационной структуры, показанной на рис. 2.28, которая называется *одноуровневым деревом*. В Паскале эта информация может храниться в одной переменной типа Record (запись). Задать тип и описать соответствующую переменную можно следующим образом:

```
Type Anketa1 = Record
                 FIO : String[50];    {Поля}
                 Pol : Char;
                 Dat : String[16];    {записи}
                 Adres : String[50];
                 Curs : 1..5;         {или элементы}
                 Grup : 1..10;
                 Stip : Real          {записи}
               End;
Var Student : Anketa1;
```

Такая запись, как и соответствующее ей дерево, называется одноуровневой.

К каждому элементу записи можно обратиться, используя *составное имя*, которое имеет следующую структуру:

```
<имя переменной>.<имя поля>
```

Например, Student.FIO; Student.Dat и т.д.

Если, например, полю «Курс» требуется присвоить значение 3, то это делается следующим образом:

```
Student.Curs := 3;
```

Поля записи могут иметь любой тип, в частности, сами могут быть записями. Такая возможность используется в случае, если

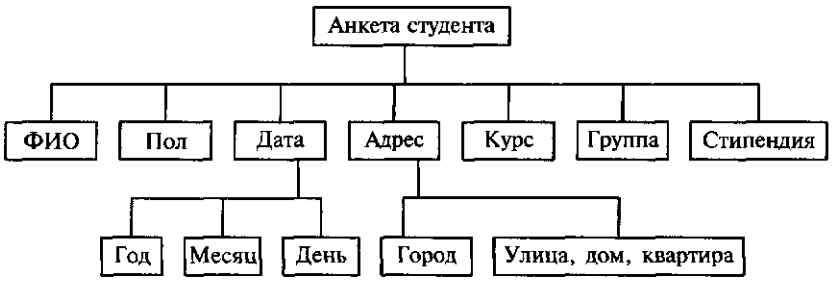


Рис. 2.29. Двухуровневое дерево

требуется представить многоуровневое дерево (более одного уровня). Например, те же сведения о студентах можно отобразить двухуровневым деревом, показанным на рис. 2.29.

Такая организация данных позволит, например, делать выборки информации по году рождения или по городу, где живут студенты. В этом случае описание соответствующей записи будет иметь следующий вид:

```
Type Anketa2 = Record
                 FIO: String[50];
                 Pol: Char;
                 Dat: Record
                      God: Integer;
                      Mes: String[10];
                      Den: 1..31
                      End;
                 Adres: Record
                        Gorod: String[20];
                        UlDomKv: String[30];
                      End;
                 Curs: 1..5;
                 Grup: 1..10;
                 Stip: Real
               End;
Var Student: Anketa2;
```

Поля такой записи, находящиеся на втором уровне, идентифицируются тройным составным именем. Например,

```
Student.Dat.God; Student.Adres.Gorod.
```

Структура описания комбинированного типа показана на рис. 2.30.

В программе могут использоваться массивы записей. Если на факультете 500 студентов, то все анкетные данные о них можно представить в массиве:

```
Var Student : Array[1..500] Of Anketa1;
```

В этом случае, например, год рождения 5-го в списке студента хранится в переменной Student[5].Dat.God.

Любая обработка записей (в том числе ввод и вывод) производится поэлементно. Например, ввод сведений о 500 студентах можно выполнить следующим образом:

```
For I := 1 To 500 Do
With Student[I] Do
Begin
    Write('ФИО: '); ReadLn(FIO);
    Write('Пол (м/ж): '); ReadLn(Pol);
    Write('Дата рождения: '); ReadLn(Dat);
    Write('Адрес: '); ReadLn(Adres);
    Write('Курс: '); ReadLn(Curs);
    Write('Группа: '); ReadLn(Grup);
    Write('Стипендия (руб.): '); ReadLn(Stip)
End;
```

В этом примере использован оператор присоединения, имеющий следующий вид:

```
With <переменная типа запись> Do <оператор>;
```

Этот оператор позволяет, один раз указав имя переменной типа «Запись» после слова With, работать в пределах оператора с именами полей как с обычными переменными, т.е. не писать громоздких составных имен.

Тип «Запись» в Паскале может иметь переменный состав полей, который изменяется в ходе выполнения программы. Эта воз-

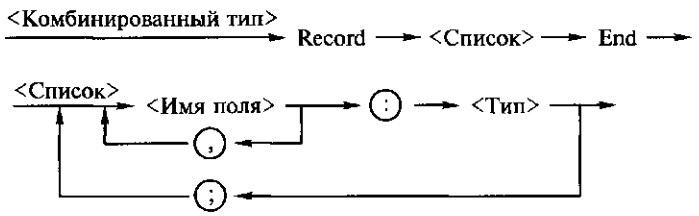


Рис. 2.30. Структура описания комбинированного типа

можность реализуется с использованием так называемой вариантной части записи (подробнее см. в специальной литературе по языку Паскаль).

Работа с файлами записей. Чаще всего записи используются как элементы файлов, составляющих компьютерные информационные системы. Рассмотрим примеры программ, работающих с файлами записей.

Пример 2.38. Требуется сформировать файл FM.DAT, содержащий экзаменационную ведомость одной студенческой группы. Записи файла состоят из следующих элементов: ФИО; номер зачетной книжки; оценка.

Программа решения задачи:

```
Program Examen;
Type Stud = Record
              FIO : String[30];
              Nz : String[6];
              Mark : 2..5
            End;
Var Fstud : File Of Stud;
  S : Stud;
  N, I : Byte;
Begin
  Assign(Fstud, 'FM.DAT'); Rewrite(Fstud);
  Write('Количество студентов в группе? ');
  ReadLn(N);
  For I := 1 To N Do
  Begin
       Write(I:1,'-й, ФИО '); ReadLn(S.FIO);
       Write('Номер зачетки: '); ReadLn(S.Nz);
       Write('Оценка: '); ReadLn(S.Mark);
       Write(Fstud, S)
  End;
  WriteLn('Формирование файла закончено!');
  Close(Fstud)
End.
```

Прежде чем перейти к следующему примеру, связанному с обработкой сформированного файла, рассмотрим еще одно средство работы с файлами.

Прямой доступ к записям файла. В стандартном языке Паскаль допустим только последовательный доступ к элементам файла. Одной из дополнительных возможностей, реализованных в Турбо Паскале, является прямой доступ к записям файла.

Как уже отмечалось, элементы файла пронумерованы в порядке их занесения в файл, начиная с нуля. Задав номер элемента

файла, можно непосредственно установить на него указатель, после чего можно считывать или перезаписывать данный элемент.

Установка указателя на заданный элемент файла производится процедурой

```
Seek(FV, n).
```

Здесь FV — имя файловой переменной, n — порядковый номер элемента.

Приведем пример использования этой процедуры.

Пример 2.39. Имеется файл, сформированный программой из примера 2.38. Пусть некоторые студенты пересдали экзамен и получили новые оценки. Требуется составить программу внесения результатов переэкзаменовки в файл, которая будет запрашивать номер студента в ведомости и его новую оценку. Работа заканчивается при вводе несуществующего номера (9 999).

Программа решения следующая:

```
Program New_Marks;
Type Stud = Record
              FIO : String[30]; Nz : String[6];
              Mark : 2..5
            End;
Var Fstud: File Of Stud;
    S : Stud;
    N : Integer;
Begin
  Assign(Fstud, 'FM.DAT');
  Reset(Fstud);
  Write('Номер в ведомости? ');
  ReadLn(N);
  While N <> 9999 Do
  Begin
       Seek(Fstud, N-1);
       Read(Fstud, S);
       Write(S.FIO, ' оценка? ');
       ReadLn(S.Mark);
       Seek(Fstud, N-1);
       Write(Fstud, S);
       Write('Номер в ведомости? ');
       ReadLn(N);
  End;
  WriteLn('Работа закончена!');
  Close(Fstud)
End.
```

Данный пример требует некоторых пояснений. Список студентов в ведомости пронумерован, начиная с единицы, а записи в

файле нумеруются с нуля. Следовательно, если n — это номер в ведомости, то номер соответствующей записи в файле равен $n - 1$. После прочтения записи номер $n - 1$ указатель смещается к следующей n-й записи. Для повторного внесения на то же место исправленной записи повторяют установку указателя.

УПРАЖНЕНИЯ

1. Описать запись, содержащую сведения о рейсе самолета.
2. Описать массив записей, содержащий таблицу химических элементов Д. И. Менделеева. Составить программу заполнения массива.
3. Рассматривая комплексное число как двухэлементную запись, составить процедуры выполнения арифметических операций с комплексными числами.
4. Сведения о деталях, хранящихся на складе, содержат следующие атрибуты: название, количество, стоимость одной детали. Составить программы, решающие следующие задачи:
   а) заполнить файл с информацией о деталях на складе;
   б) вычислить общую стоимость деталей;
   в) выяснить, какие детали имеются в наибольшем количестве и какие — в наименьшем;
   г) вывести информацию о наличии на складе деталей данного типа и об их количестве;
   д) внести изменения в файл после выдачи со склада определенного количества данного вида деталей. Если какой-то тип деталей полностью выбран со склада, уничтожить запись о нем в файле.

2.21. УКАЗАТЕЛИ И ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Ранее рассматривалось программирование, связанное с обработкой только статических данных.

Статическими называются величины, память под которые выделяется во время компиляции и сохраняется в течение всей работы программы.

В Паскале существует и другой способ выделения памяти под данные — динамический. *Динамическими* называются величины, память под которые отводится во время выполнения программы.

Раздел оперативной памяти, распределяемый статически, называется *статической* памятью; динамически распределяемый раздел памяти называется *динамической* памятью.

Использование динамических величин предоставляет программисту ряд дополнительных возможностей. Во-первых, подключе-

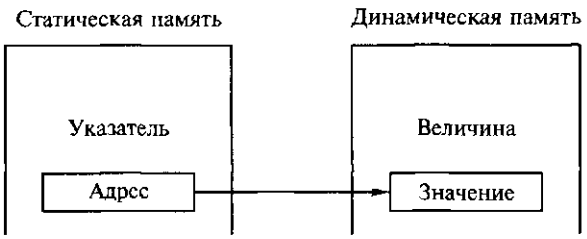


Рис. 2.31. Схема действия указателя

ние динамической памяти позволяет увеличить объем обрабатываемых данных. Во-вторых, если потребность в каких-то данных отпала до окончания программы, то занятую ими память можно освободить для другой информации. В-третьих, использование динамической памяти позволяет создавать структуры данных переменного размера.

Работа с динамическими величинами связана с использованием еще одного типа данных — ссылочного.

Указатели. Величины, имеющие ссылочный тип, называют *указателями*.

Указатель содержит в динамической памяти адрес поля, в котором хранится величина определенного типа. Сам указатель располагается в статической памяти (рис. 2.31).

Адрес — это номер первого байта поля памяти, в котором располагается величина. Размер поля однозначно определяется типом.

Величина ссылочного типа (указатель) описывается в разделе описания переменных следующим образом:

```
Var <идентификатор> : <ссылочный тип>;
```

В стандартном Паскале каждый указатель может ссылаться на величину только одного определенного типа, который называется *базовым* для данного указателя. Имя базового типа указывается в описании в следующей форме:

```
<ссылочный тип> := ^<имя типа>;
```

Приведем примеры описания указателей:

```
Type  Massiv = Array[1..100] Of Integer;
Var   P1 : ^Integer;
      P2 : ^Char;
      PM : ^Massiv;
```

Здесь *P1* — указатель на динамическую величину целого типа; *P2* — указатель на динамическую величину символьного типа;

PM — указатель на динамический массив, тип которого задан в разделе Type.

Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции память выделяется только под статические величины. Указатели — это статические величины, поэтому они требуют описания.

Память под динамическую величину, связанную с указателем, выделяется в результате выполнения стандартной процедуры NEW. Формат обращения к этой процедуре следующий:

```
NEW(<указатель>);
```

Считается, что выполнение этого оператора создает динамическую величину, имя которой имеет следующий вид:

```
<имя динамической величины> ::= <указатель>^.
```

Пусть в программе, в которой имеется приведенное ранее описание указателя, имеются следующие операторы:

```
NEW(P1); NEW(P2); NEW(PM);
```

После их выполнения в динамической памяти оказывается выделенным место под три величины (две скалярные и один массив), имеющие соответственно идентификаторы *P*1^ пт, *P*2^ пт, *PM*^. Например, *P*1^ — это обозначение динамической переменной, на которую ссылается указатель *P*1.

Связь между динамическими величинами и их указателями показана на схеме, представленной на рис. 2.32.

Далее работа с динамическими переменными происходит точно так же, как со статическими переменными соответствующих типов, т.е. им можно присваивать значения, их можно использовать в качестве операндов в выражениях, параметров подпрограмм и т.д. Например, если переменной *P*1^ требуется присвоить число 25, переменной *P*2^ присвоить значение символа «W», а массив *PM*^ заполнить по порядку целыми числами от 1 до 100, это делается следующим образом:

```
P1^ := 25; P2^ := 'W';
For I := 1 To 100 Do PM^[I] := I;
```

Кроме процедуры NEW значение указателя может определяться следующим оператором присваивания:

```
<указатель> := <ссылочное выражение>;
```

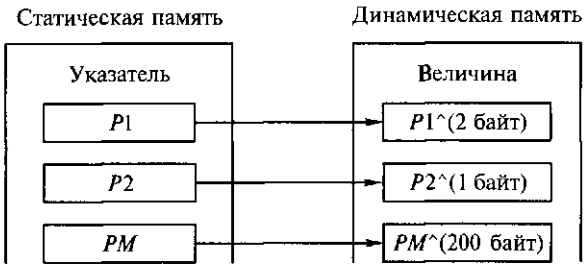


Рис. 2.32. Связь указателей с динамическими величинами

В качестве ссылочного выражения можно использовать:

- указатель;
- ссылочную функцию (т. е. функцию, значением которой является указатель);
- константу Nil.

Nil — это зарезервированная константа, обозначающая пустую ссылку, т. е. ссылку, которая ни на что не указывает. При присваивании базовые типы указателя и ссылочного выражения должны быть одинаковыми. Константу Nil можно присваивать указателю с любым базовым типом.

Ссылочная переменная до присваивания ей значения (с помощью оператора присваивания или процедуры NEW) является неопределенной.

Ввод и вывод указателей не допускается.

Рассмотрим пример. Пусть в программе описаны следующие указатели:

```
Var D, P : ^Integer;
    K : ^Boolean;
```

Тогда допустимыми являются операторы присваивания

```
D := P; K := Nil,
```

поскольку должен соблюдаться принцип соответствия типов. Оператор $K := D$ ошибочен, так как базовые типы правой и левой частей разные.

Если динамическая величина теряет свой указатель, она становится «мусором». В программировании так называют не нужную информацию, занимающую память.

Представьте, что в программе с описанными ранее указателями в разделе операторов записано следующее:

```
NEW(D); NEW(P);
{Выделено место в динамической памяти под две це-
лые переменные. Указатели получили соответствующие
значения}
D^ := 3; P^ := 5;
{Динамическим переменным присвоены значения}
P := D;
{Указатели P и D стали ссылаться на одну и ту же
величину, равную 3}
WriteLn(P^, D^); {Дважды печатается число 3}
```

Таким образом, динамическая величина, равная 5, потеряла свой указатель и стала недоступной. Однако место в памяти она занимает. Это и есть пример возникновения «мусора». Что произошло в результате выполнения оператора $P := D$ показано на рис. 2.33.

В Паскале имеется стандартная процедура, позволяющая освобождать память от данных, потребность в которых отпала. Ее формат

```
DISPOSE(<указатель>);
```

Например, если динамическая переменная $P^$ больше не требуется, оператор DISPOSE(P) удалит ее из памяти, после чего значение указателя P станет неопределенным. Особенно существенный эффект экономии памяти получают при удалении больших массивов данных.

В версиях Турбо Паскаля, работающих с операционной системой MS DOS, под данные одной программы выделяется 64 Кбайт памяти. Это и есть статическая область памяти. При необходимости работы с большими массивами информации этого может оказаться недостаточно. Размер динамической памяти намного больше (сотни килобайт), поэтому ее использование позволяет существенно увеличить объем обрабатываемой информации.

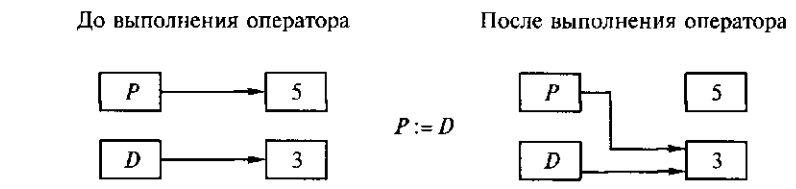


Рис. 2.33. Механизм возникновения «мусора»

Следует отчетливо понимать, что работа с динамическими данными замедляет выполнение программы, поскольку доступ к величине происходит в два шага: сначала определяется указатель, а затем по нему — величина. Как это часто бывает, действует «закон сохранения неприятностей», т.е. выигрыш в объеме памяти компенсируется проигрышем во времени.

Пример 2.40. Требуется создать вещественный массив из 10 000 чисел и заполнить его случайными числами в диапазоне от 0 до 1. Вычислить среднее значение массива. Очистить динамическую память. Создать целый массив размером 10 000, заполнить его случайными целыми числами в диапазоне от –100 до 100 и вычислить его среднее значение.

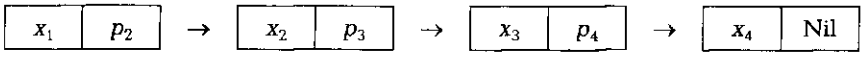
Программа решения следующая:

```
Program Sr;
Const NMax = 10000;
Type Diapazon = 1..NMax;
     MasInt = Array[Diapazon] Of Integer;
     MasReal = Array[Diapazon] Of Real;
Var PInt : ^MasInt;
    PReal : ^MasReal;
    I, Midint : longInt;
    MidReal : Real;
Begin
  MidReal := 0; MidInt := 0;
  Randomize;
  NEW(PReal);
  {Выделение памяти под вещественный массив}
  {Вычисление и суммирование массива}
  For I := 1 To NMax Do
  Begin PReal^[I] := Random;
        MidReal := MidReal + PReal^[I]
  End;
  DISPOSE(PReal);{Удаление вещественного массива}
  NEW(PInt); {Выделение памяти под целый массив}
  {Вычисление и суммирование целого массива}
  For I := 1 To NMax Do
  Begin
        PInt^[I] := Random(200) - 100;
        MidInt := MidInt + PInt^[I]
  End;
  {Вывод средних значений}
  WriteLn('Среднее целое равно:',MidInt Div Max);
  WriteLn('Среднее вещественное равно: ',
  (MidReal / NMax):10:6)
End.
```

Связанные списки. Рассмотрим теперь, как в динамической памяти можно создать структуру данных переменного размера.

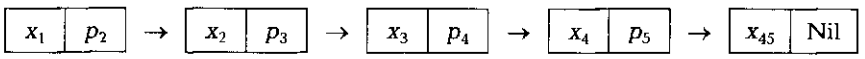
Разберем следующий пример. В процессе физического эксперимента многократно снимаются показания прибора (допустим, термометра) и записываются в компьютерную память для дальнейшей обработки. Заранее неизвестно, сколько будет произведено измерений.

Если для обработки таких данных не использовать внешнюю память (файлы), то разумно расположить их в динамической памяти. Во-первых, динамическая память позволяет хранить больший объем информации, чем статическая. Во-вторых, в динамической памяти эти числа можно организовать в *связанный список*, который не требует предварительного указания количества чисел подобно массиву. Что же такое связанный список? Схематически его можно представить следующим образом:



Каждый элемент списка состоит из двух частей: информационной (x_1, x_2, ...) и указателя на следующий элемент (p_2, p_3, ...). Последний элемент имеет пустой указатель Nil. Связанный список такого типа называется *однонаправленной цепочкой*.

В сформулированной задаче информационная часть содержит следующие вещественные числа: x_1 — результат первого измерения; x_2 — результат второго измерения; x_3 — результат третьего измерения; x_4 — результат последнего измерения. Связанный список обладает тем замечательным свойством, что его можно дополнять по мере поступления новой информации. Дополнение происходит присоединением нового элемента к концу списка, а значение Nil в последнем элементе заменяется ссылкой на новый элемент цепочки:



Связанный список не занимает лишней памяти. Память расходуется в том объеме, который требуется для поступившей информации.

В программе для представления элементов цепочки используется комбинированный тип данных — запись. В нашем примере тип такого элемента может быть следующим:

```
Type Pe = ^Elem;
     Elem = Record
```

```
    T : Real;
    P : Pe
End;
```

Здесь Pe — это ссылочный тип на переменную типа Elem. Этим именем обозначен комбинированный тип, состоящий из двух полей: T — вещественной величины, хранящей значение температуры, и P — указателя на динамическую величину типа Elem.

В данном описании нарушен один из основных принципов языка Паскаль, согласно которому на любой программный объект можно ссылаться только после его описания. В самом деле, тип Pe определяется через тип Elem, который, в свою очередь, определяется через тип Pe. Однако в Паскале существует единственное исключение из этого правила, связанное со ссылочным типом. Приведенный фрагмент программы является правильным.

Пример 2.41. Программа формирования связанного списка в ходе ввода данных:

```
Type Pe = ^TypElem;
     TypElem = Record
                 T : Real; P : Pe
               End;
Var  Elem, Beg : Pe;
     X : Real; Ch : Char;
Begin
{Определение адреса начала списка и его сохранение}
  NEW(Elem);
  Beg := Elem;
  Elem^.P := Elem;
{Диалоговый ввод значений с занесением их в список
и организацией связи между элементами}
While Elem^.P <> Nil Do
Begin
  Write('Вводите число: '); ReadLn(Elem^.T);
  Write('Повторить ввод? (Y/N)'); ReadLn(Ch);
  If (Ch = 'n') Or (Ch = 'N')
  Then Elem^.P := Nil
  Else Begin
         NEW(Elem^.P);
         Elem := Elem^.P
       End
End;
WriteLn('Ввод данных закончен');
{Вывод полученной числовой последовательности}
WriteLn('Контрольная распечатка');
```

```
Elem := Beg;
Repeat
  WriteLn(N, ':', Elem^.T : 8 : 3);
  Elem := Elem^.P
Until Elem = Nil
End.
```

Здесь ссылочная переменная Beg используется для сохранения адреса начала цепочки. Всякая обработка цепочки начинается с ее первого элемента. В программе показано, как происходит продвижение по цепочке при ее обработке (в данном примере при распечатке информационной части списка по порядку).

Однонаправленная цепочка — простейший вариант связанного списка. В практике программирования используются также двунаправленные цепочки (когда каждый элемент хранит указатель на следующий и предыдущий элементы списка) и двоичные деревья. Подобные структуры данных называются *динамическими*.

Одной из часто употребляемых в программировании динамических структур данных является стек. *Стек* — это связанная цепочка, начало которой называется вершиной и состав элементов которой постоянно меняется. Каждый вновь поступающий элемент устанавливается в вершине стека. Удаление элементов из стека также производится с вершины.

Стек подобен детской пирамидке, в которой на стержень надеваются кольца. Всякое новое кольцо оказывается на ее вершине. Снимать кольца можно только сверху. Принцип изменения содержания стека часто формулируют как «последним пришел — первым вышел».

Пример 2.42. Требуется составить процедуры добавления элемента в стек (INSTEK) и исключения элемента из стека (OUTSTEK), если эти элементы — символьные величины.

В процедурах используется тип данных Pe, который должен быть глобально объявлен в основной программе:

```
Type Pe = ^TypElem;
     TypElem = Record
                 C : Char; P : Pe
               End;
```

В приведенной далее процедуре INSTEK аргументом является параметр Sim, содержащий включаемый в стек символ. Ссылочная переменная Beg содержит указатель на вершину стека при входе в процедуру и при выходе из нее. Следовательно, этот параметр яв-

ляется и аргументом, и результатом процедуры. В случае когда стек пустой, указатель Beg равен Nil.

```
Procedure INSTEK(Var Beg : Pe; Sim : Char);
Var X : Pe;
Begin New(X);
      X^.C := Sim;
      X^.P := Beg;
      Beg := X
  End;
```

В приведенной далее процедуре OUTSTEK параметр Beg играет ту же роль, что и в предыдущей процедуре, т. е. после удаления последнего символа его значение станет равным Nil. Ясно, что если стек пустой, то удалять из него нечего. Логический параметр Flag позволяет распознать этот случай: если на выходе из процедуры Flag = True, это значит, что удаление произошло, а если Flag = = False, это значит, что стек был пуст и ничего не изменилось.

```
Procedure OUTSTEK(Var Beg : Pe; Var Flag : Boolean);
Var X : Pe;
Begin
  If Beg = Nil
  Then Flag := False
  Else Begin
         Flag := True;
         X := Beg;
         Beg := Beg^.P;
         DISPOSE(X)
       End
End;
```

Данная процедура не оставляет после себя «мусора», освобождая память от удаленных элементов.

УПРАЖНЕНИЯ

1. В программе имеются описания.

```
Var P, Q : ^Integer; R : ^Char;
```

Определить, какие из следующих операторов неправильные и почему:

а) `P := Q;` б) `Q := R;` в) `P := Nil;` г) `R := Nil;`

д) `Q := P^;` е) `P^ := Nil;` ж) `R^: = P^;` з) `Q^ := Ord(R^);`
и) **If** `R <> Nil` **Then** `R^ := Nil^;` к) **If** `Q > Nil` **Then** `Q^ := P^;`
л) **If** `P = Q` **Then** `Write(Q);` м) **If** `Q <> R` **Then** `Read(R^).`

2. В программе имеются описания

```
Type A = ^Char;
     B = Record F1 : Char; F2 : A End;
Var P : ^B; Q: A;
```

Определить значения всех указателей и динамических переменных после выполнения следующих операторов:

```
NEW(Q); Q^ := '7';
NEW(P); P^.F1 := Succ(Q^); P^.F2 := Q;
```

3. Найти ошибки в следующей программе:

```
Program Example;
Var A, B : ^Integer;
Begin If A = Nil Then Read(A);
  A^ := 5; B := Nil; B^ := 2;
  NEW(B); Read(B^); WriteLn(B, B^);
  NEW(A); B := A; DISPOSE(A); B^:= 4
End.
```

4. Составить программу занесения в динамическую память вещественного массива из 10 000 чисел, хранящегося в файле на магнитном диске, а также поиска в нем значения и номера первого максимального элемента.

5. Выполнить четвертое упражнение при условии, что размер числового массива заранее не определен (определяется размером файла: предполагается, что размер файла не превышает размера динамической памяти). Данные в динамической памяти организовать в виде связанного списка.

6. Связанный список представляет собой символьную цепочку из строчных латинских букв. Описать следующую процедуру или функцию:

а) определяющую, является ли список пустым;
б) заменяющую в списке все вхождения одного символа на вхождения другого;
в) меняющую местами первый и последний элементы непустого списка;
г) проверяющую, упорядочены ли элементы списка по алфавиту;
д) определяющую, какая буква входит в список наибольшее количество раз.

7. Составить тестовую программу, проверяющую правильность работы процедур INSTEK и OUTSTEK из примера 2.42.

8. Очередью называется динамическая структура (связанная цепочка), в которой действует принцип «первым пришел — первым вышел». Написать процедуры включения элемента в очередь и исключения элемента из очереди. Составить тестовую программу, проверяющую работу процедур.

2.22. ВНЕШНИЕ ПОДПРОГРАММЫ И МОДУЛИ

Стандартный Паскаль не располагает средствами разработки и поддержки библиотек программиста (в отличие от Фортрана и других языков программирования высокого уровня), которые компилируются отдельно и в дальнейшем могут быть использованы не только самим разработчиком. Если программист имеет достаточно большие наработки и если те или иные подпрограммы могут быть использованы при написании новых приложений, приходится эти подпрограммы целиком включать в новый текст.

В Турбо Паскале это ограничение преодолевается, во-первых, введением внешних подпрограмм, а во-вторых, созданием и использованием модулей. Рассмотрим оба способа.

Введение внешних подпрограмм. В этом случае исходный текст каждой процедуры или функции хранится в отдельном файле и при необходимости с помощью специальной директивы компилятора включается в текст создаваемой программы.

Проиллюстрируем этот прием на примере следующей задачи целочисленной арифметики: дано натуральное число n, требуется найти сумму первой и последней цифр этого числа.

Для решения задачи используем функцию, вычисляющую число цифр в записи натурального числа. Например:

```
Function Digits(N : LongInt): Byte;
Var Kol : Byte;
Begin
    Kol := 0;
    While N <> 0 Do
    Begin
          Kol := Kol + 1;
          N := N Div 10
    End;
    Digits := Kol
End;
```

Сохраним этот текст в файле с расширением.inc (расширение внешних подпрограмм в Турбо Паскале), например digits.inc.

Опишем также функцию возведения натурального числа в натуральную степень (a^n):

```
Function Power(A,N:LongInt):LongInt;{файл power.inc}
Var I, St : LongInt;
Begin
```

```
  St := 1;
  For  I := 1 To N Do
       St := St * A;
       Power := St
End;
```

Теперь составим основную программу, решающую поставленную задачу с использованием описанных функций:

```
Program Example1;
Var N, S : Integer;
{$I digits.inc} {Подключение внешней функции из
файла digits.inc, вычисляющей количество цифр в
записи числа}
{$I power.inc} {Подключение внешней функции из
файла power.inc, вычисляющей результат возведения
числа A в степень N}
Begin
  Write('Введите натуральное число: '); ReadLn(N);
{Для определения последней цифры числа N берется
остаток от деления этого числа на 10, а для опре-
деления первой цифры число N делится на число 10,
возведенное в степень, на единицу меньшую, чем ко-
личество цифр в записи числа N (так как нумерация
разрядов начинается с 0)}
  S := N Mod 10 + N Div Power(10, Digits(N) - 1);
  WriteLn('Искомая сумма: ', S)
End.
```

Директива компилятора {$I <имя файла>} позволяет в данное место текста программы вставить содержимое файла с указанным именем.

Файлы с расширением.inc можно накапливать на магнитном диске, формируя таким образом личную библиотеку подпрограмм.

Внешние процедуры создаются и внедряются в использующие их программы так же, как и функции в рассмотренной задаче.

Создание и использование модулей. Рассмотрим теперь структуру, разработку, компиляцию и использование модулей.

Модуль — это набор ресурсов (функций, процедур, констант, переменных, типов и т.д.), разрабатываемых и хранимых независимо от использующих их программ. В отличие от внешних подпрограмм модуль может содержать достаточно большой набор

процедур и функций, а также других ресурсов для разработки программ. В основе идеи модульности лежат принципы структурного программирования. Существуют стандартные модули Турбо Паскаля (SYSTEM, CRT, GRAPH и др.), справочная информация по которым дана в приложениях 1...3.

Модуль имеет следующую структуру:

```
Unit <имя модуля>; {Заголовок модуля}
Interface
  {Интерфейсная часть}
Implementation
  {Раздел реализации}
Begin
  {Раздел инициализации модуля}
End.
```

После служебного слова Unit записывается имя модуля, которое (для удобства дальнейших действий) должно совпадать с именем файла, содержащего данный модуль. Следовательно (как принято в MS DOS), имя не должно содержать более восьми символов.

В разделе Interface объявляются все ресурсы, которые будут в дальнейшем доступны программисту при подключении модуля. Для подпрограмм здесь указывается лишь полный заголовок.

В разделе Implementation описываются все подпрограммы, которые были ранее объявлены. Кроме того, в нем могут содержаться свои константы, переменные, типы, подпрограммы, которые носят вспомогательный характер и используются для написания основных подпрограмм. В отличие от ресурсов, объявленных в разделе Interface, все, что дополнительно объявляется в Implementation, уже не будет доступно при подключении модуля. При описании основной подпрограммы достаточно указать ее имя, не переписывая полностью весь заголовок, а затем записать тело.

Раздел инициализации (зачастую отсутствующей) содержит операторы, которые должны быть выполнены сразу же после запуска программы, использующей модуль.

Приведем пример разработки и использования модуля. (Поскольку рассматриваемая далее задача достаточно элементарна, ограничимся распечаткой текста программы с подробными комментариями.)

Пример 2.43. Реализовать в виде модуля набор подпрограмм для выполнения над обыкновенными дробями вида P/Q (где P — целое число, Q — натуральное число) следующих операций: сложения, вычитания, умножения,

деления, сокращения, возведения в степень N (где N — натуральное число), а также функций, реализующих операции отношения (равно, не равно, больше или равно, меньше или равно, больше, меньше).

При этом дробь представить следующим типом:

```
Type Frac = Record
              P : Integer;
              Q : 1..32767
            End;
```

С использованием разработанного модуля решить две задачи.

1. Дан массив A, элементы которого — обыкновенные дроби. Найти сумму всех элементов и их среднее арифметическое. Ответы представить в виде несократимых дробей.

2. Дан массив A, элементы которого — обыкновенные дроби. Отсортировать его в порядке возрастания.

Реализация набора подпрограмм в виде модуля:

```
Unit Droby;
Interface
 Type
    Natur = 1..High(LongInt);
    Frac = Record
             P : LongInt;
             Q : Natur
           End;
Procedure Sokr(Var A : Frac);
Procedure Summa(A, B : Frac; Var C : Frac);
Procedure Raznost(A, B : Frac; Var C : Frac);
Procedure Proizvedenie(A, B : Frac; Var C : Frac);
Procedure Chastnoe(A, B: Frac; Var C : Frac);
Procedure Stepen(A : Frac; N : Natur; Var C : Frac);
Function Menshe(A, B : Frac): Boolean;
Function Bolshe(A, B : Frac): Boolean;
Function Ravno(A, B : Frac): Boolean;
Function MensheRavno(A, B : Frac): Boolean;
Function BolsheRavno(A, B : Frac): Boolean;
Function NeRavno(A, B : Frac): Boolean;
{Раздел реализации модуля}
Implementation
{Наибольший общий делитель двух чисел -
вспомогательная функция, ранее не объявленная}
Function NodEvklid(A, B : Natur): Natur;
Begin
  While A <> B Do
    If A > B Then
       If A Mod B <> 0  Then A := A Mod B
                        Else A := B
```

```
        Else
        If B Mod A <> 0  Then B := B Mod A
                         Else B := A;
     NodEvklid := A
End;
{Сокращение дроби}
Procedure Sokr;
Var M, N : Natur;
Begin
  If A.P <> 0
  Then
  Begin
     If A.P < 0
        Then M := Abs(A.P)
        Else M := A.P;   {Совмещение типов, так как
                         A.P-LongInt}
        N := NodEvklid(M, A.Q);
        A.P := A.P Div N;
        A.Q := A.Q Div N
  End
End;
Procedure Summa; {Сумма дробей}
Begin
   {Знаменатель дроби}
   C.Q := (A.Q * B.Q) Div NodEvklid(A.Q, B.Q);
   {Числитель дроби}
   C.P := A.P * C.Q Div A.Q + B.P * C.Q Div B.Q;
   Sokr(C)
End;
Procedure Raznost; {Разность дробей}
Begin
   {Знаменатель дроби}
   C.Q := (A.Q * B.Q) Div NodEvklid(A.Q, B.Q);
   {Числитель дроби}
   C.P := A.P * C.Q Div A.Q-B.P * C.Q Div B.Q;
   Sokr(C)
End;
Procedure Proizvedenie; {Умножение дробей}
Begin
  {Знаменатель дроби}
  C.Q := A.Q * B.Q;
  {Числитель дроби}
  C.P := A.P * B.P;
  Sokr(C)
End;
Procedure Chastnoe; {Деление дробей}
Begin
```

```
  {Знаменатель дроби}
  C.Q := A.Q * B.P;
  {Числитель дроби}
  C.P := A.P * B.Q;
  Sokr(C)
End;
Procedure Stepen; {Возведение дроби в степень}
Var I : Natur;
Begin
  C.Q := 1;
  C.P := 1;
  Sokr(A);
  For I := 1 To N Do
     Proizvedenie(A, C, C)
End;
Function Menshe;{Отношение «<» между дробями}
Begin
  Menshe := A.P * B.Q < A.Q * B.P
End;
Function Bolshe; {Отношение «>» между дробями}
Begin
  Bolshe := A.P * B.Q > A.Q * B.P
End;
Function Ravno; {Отношение «=» между дробями}
Begin
  Ravno := A.P * B.Q = A.Q * B.P
End;
Function BolsheRavno; {Отношение «≥» между дробями}
Begin
  BolsheRavno := Bolshe(A, B) Or Ravno(A, B)
End;
Function MensheRavno; {Отношение «≤» между дробями}
Begin
  MensheRavno := Menshe(A, B) Or Ravno(A, B)
End;
Function NeRavno; {Отношение «<>» между дробями}
Begin
  NeRavno := Not Ravno(A, B)
End;
{Раздел инициализации модуля}
Begin
End.
```

При разработке модуля рекомендуется следующий порядок:

1) спроектировать модуль, т. е. определить основные и вспомогательные подпрограммы и другие ресурсы;

2) описать компоненты модуля;

3) отладить каждую подпрограмму отдельно, после чего «вклеить» в текст модуля.

Сохраним текст разработанной программы в файле DROBY.PAS и откомпилируем модуль, используя внешний компилятор, поставляемый вместе с Турбо Паскалем. Команда будет иметь вид TPC DROBY.PAS. Если в тексте нет синтаксических ошибок, получим файл DROBY.TPU, иначе будет выведено соответствующее сообщение с указанием строки, содержащей ошибку. Другой вариант компиляции: в меню системы программирования Турбо Паскаль выбрать Compile/Destination Disk, а затем — Compile/Build.

Теперь можно подключить разработанный модуль к программе, где планируется его использование.

Программа решения первой поставленной задачи суммирования массива дробей:

```
Program Sum;
Uses Droby;
Var A : Array[1..100] Of Frac;
    I, N : Integer;
    S : Frac;
Begin
  Write('Введите количество элементов массива: ');
  ReadLn(N);
  S.P := 0; S.Q := 1; {Первоначально сумма равна нулю}
  For I := 1 To N Do {Вводим и суммируем дроби}
  Begin
    Write('Введите числитель ', I, '-й дроби: ');
    ReadLn(A[I].P);
    Write('Введите знаменатель ', I, '-й дроби: ');
    ReadLn(A[I].Q);
    Summa(A[I], S, S);
  End;
  WriteLn('Ответ: ', S.P, '/', S.Q)
End.
```

Вторую поставленную задачу — сортировку массива — читателю предлагается решить самостоятельно.

Как видно из приведенного примера, для подключения модуля используется служебное слово Uses, после которого указывается имя модуля. Данная строка записывается сразу же после заголовка программы. Если необходимо подключить несколько модулей, они перечисляются через запятую.

При использовании ресурсов модуля программисту совсем не обязательно знать, как работают вызываемые подпрограммы. Достаточно знать назначение подпрограмм и их спецификации, т.е. имена и параметры. По этому принципу осуществляется работа со всеми стандартными модулями. Следовательно, если программист

разрабатывает модули не только для личного пользования, ему необходимо выполнить полное описание всех доступных при подключении ресурсов.

УПРАЖНЕНИЯ

1. Используя описанный в данном подразделе модуль Droby, решить следующую задачу. Дан массив *A* обыкновенных дробей. Найти сумму всех дробей и результат представить в виде несократимой дроби. Вычислить среднее арифметическое всех дробей и результат представить в виде несократимой дроби.

2. Используя описанный в данном подразделе модуль Droby, решить следующую задачу. Дан массив *A* обыкновенных дробей. Отсортировать его в порядке возрастания.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Кем, когда и с какой целью был разработан язык программирования Паскаль?
2. Из каких разделов состоит программа на Паскале (в версии Турбо Паскаль)?
3. Какие группы символов входят в алфавит Паскаля?
4. По каким правилам задаются идентификаторы? Приведите примеры верных и ошибочных вариантов идентификаторов.
5. Какие типы данных относятся к порядковым и какие общие свойства их объединяют?
6. Чем отличаются простые типы данных от структурированных? Назовите структурированные типы данных Паскаля (Турбо Паскаля).
7. Как представляются в разделе описаний типов перечисляемый и интервальный типы? Приведите примеры.
9. В каких случаях результатом арифметической операции становится величина целого типа? Приведите примеры.
10. Как реализовать в Паскале возведение в степень в случае целого и в случае вещественного значения показателя степени?
11. Какие логические операции Паскаля и правила их выполнения (таблицы истинности) вам известны?
12. Расположите в порядке убывании приоритетов все операции Паскаля из следующих групп: арифметические, логические, отношения.
13. С помощью каких операторов реализуются в программе ввод данных с клавиатуры и вывод на экран? Опишите их формат.
14. Каким способом можно управлять форматом вывода данных? Приведите примеры для данных разных типов.

15. Нарисуйте синтаксическую диаграмму оператора ветвления и сформулируйте правила его выполнения.
16. Нарисуйте синтаксическую диаграмму оператора выбора и сформулируйте правила его выполнения.
17. Нарисуйте синтаксическую диаграмму оператора цикла с предусловием и сформулируйте правила его выполнения.
18. Нарисуйте синтаксическую диаграмму оператора цикла с постусловием и сформулируйте правила его выполнения.
19. Нарисуйте синтаксическую диаграмму оператора цикла с параметром и сформулируйте правила его выполнения.
20. Подберите задачу, которую можно решить, используя три варианта операторов цикла: цикл «Пока», цикл «До», цикл с параметром. Составьте программы.
21. В чем состоит разница между функцией и процедурой? Придумайте задачу и опишите ее решение в форме функции и в форме процедуры.
22. Придумайте пример задачи, которую можно решить с помощью процедуры и нельзя решить с помощью функции.
23. Что такое формальные и фактические параметры подпрограммы? В чем заключается разница между параметрами-переменными и параметрами-значениями в их описании и использовании при обращении к подпрограмме? Проиллюстрируйте на примере.
24. Каковы правила соответствия между формальными и фактическими параметрами? Сформулируйте их.
25. В чем состоит разница между локальными и глобальными описаниями? Приведите пример передачи данных в подпрограмму через глобальные переменные.
26. Что такое символьная строка? Чем она отличается от массива символов?
27. Какие существуют в Паскале операции, функции и процедуры для работы со строками?
28. Как описываются в программе величины регулярного типа (массивы)?
29. Что такое размер массива и размерность массива?
30. Как идентифицируются элементы массива?
31. В какой последовательности в памяти компьютера располагаются элементы многомерного массива?
32. Какие действия можно выполнять над массивом в целом (не поэлементно)?
33. Приведите фрагмент программы построчного вывода матрицы (двумерного массива) на экран компьютера.
34. Что такое внешний и внутренний файлы?
35. С помощью какого оператора устанавливается соответствие между файловой переменной (внутренним файлом) и внешним файлом? Опишите его формат.

36. С помощью каких операторов производится открытие файла для чтения и записи?
37. Что происходит при выполнении оператора закрытия файла?
38. В чем заключается разница между типизированными файлами и текстовыми файлами?
39. С помощью каких операторов осуществляются ввод и вывод в типизированные и текстовые файлы? Приведите примеры.
40. В чем состоит различие в использовании файлов последовательного и прямого доступа?
41. Что такое комбинированный тип данных и как он описывается в программе? Как описывается величина комбинированного типа (запись)? Приведите пример.
42. Как идентифицируются элементы записи? Приведите примеры.
43. В чем заключается различие между статическими и динамическими величинами, статической и динамической памятью?
44. Что такое указатель и как он описывается в программе? Как идентифицируются динамические переменные?
45. С помощью какого оператора происходит выделение места в памяти под динамическую величину?
46. С помощью какого оператора происходит освобождение места в памяти, ранее выделенного под динамическую величину?
47. Что такое связанный список? Как он формируется? Что такое стек?
48. В чем состоит разница между внутренней и внешней подпрограммой? Какие существуют способы создания внешних подпрограмм?
49. Что такое модуль? Какую структуру он имеет?
50. Что нужно сделать в программе для того, чтобы в ней можно было использовать ресурсы модуля?

Глава 3

МЕТОДЫ ПОСТРОЕНИЯ АЛГОРИТМОВ

Из данной главы вы узнаете:

- о реализации на языке программирования Паскаль технологии последовательной детализации при составлении сложной программы;
- об отладке программы и построении полной системы тестов;
- как программируется задача о Ханойской башне с помощью рекурсивной процедуры;
- что такое задачи перебора и как можно оптимизировать полный перебор всех вариантов;
- о рекурсивном алгоритме перебора с возвратом на примере задачи о прохождении лабиринта;
- о критериях сложности алгоритмов;
- о постановке задачи сортировки данных;
- что такое алгоритм сортировки включением;
- какова сущность алгоритма быстрой сортировки Э. Хоара;
- об оценках сложности алгоритмов.

Вы научитесь:

- применять метод последовательной детализации при программировании сложных задач;
- использовать рекурсивные алгоритмы для решения комбинаторных задач;
- оценивать сложность алгоритмов.

3.1. МЕТОД ПОСЛЕДОВАТЕЛЬНОЙ ДЕТАЛИЗАЦИИ

В соответствии с основами структурного программирования, метод последовательной детализации является приемом проектирования сложных алгоритмов. Суть этого метода (см. подразд. 1.6) заключается в следующем: сначала анализируется исходная задача, в ней выделяются подзадачи и строится иерархия таких подзадач (рис. 3.1). Далее составляются алгоритмы (или программы), начиная с основного алгоритма (основной программы), а затем — вспомогательные алгоритмы с последовательным углублением уровня, пока не будут получены алгоритмы (подпрограммы), состоящие из простых команд.

Вернемся к примеру 2.17, в котором приведена программа Interpretator для решения исходной символьной строки вида '*A* + *B* ='.

Сформулируем требования к программе Interpretator, которые сделают ее более универсальной, чем вариант, рассмотренный в подразд. 2.15.

1. Операнды *a* и *b* могут быть многозначными целыми положительными числами в пределах MaxInt.

2. Между элементами строки, а также в ее начале и конце могут быть пробелы.

3. Должен осуществляться синтаксический контроль текста, который ограничивается простейшим вариантом: строка должна состоять только из цифр, знаков операций, знака «=» и пробелов.

4. Должен проводиться семантический контроль: строка должна быть построена по схеме '*a* ⊕ *b* ='. Ошибка, если какой-то элемент отсутствует или нарушен их порядок.

5. Должен осуществляться контроль диапазона значений операндов и результата (значения не должны выходить за пределы MaxInt).

Из приведенного перечня требований ясно, что программа будет непростой. Составлять ее будем, используя метод последовательной детализации. Начнем с представления в самом общем виде алгоритма как линейной последовательности этапов решения задачи.

1. Ввод строки.

2. Синтаксический контроль (определение, нет ли недопустимых символов).

3. Семантический контроль (определение, правильно ли построено выражение).

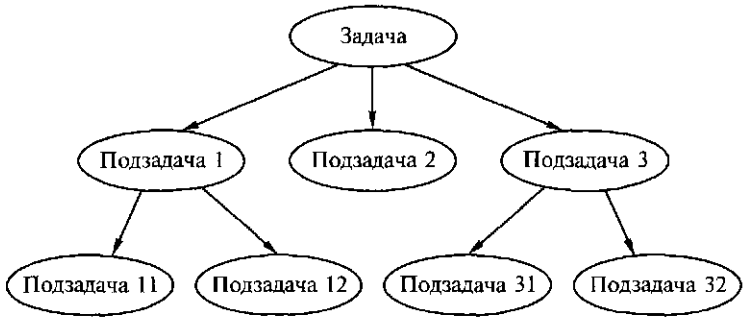


Рис. 3.1. Иерархия подзадач

4. Выделение операндов, проверка их соответствия допустимому диапазону значений и перевод в целые числа.

5. Выполнение операции, проверка соответствия результата допустимому диапазону значений.

6. Вывод результата.

Этапы 2...5 будем рассматривать как подзадачи первого уровня, соответственно назвав их (и будущие подпрограммы) Sintax, Semantika, Operand, Calc. Для их реализации потребуется решение следующих подзадач: пропуск лишних пробелов (Propusk) и преобразование символьной цифры в целое число (Cifra). Кроме того, при выделении операндов потребуется распознавать операнд, превышающий максимально допустимое значение (Error). Обобщая все сказанное в схематической форме, получаем некоторую структуру подзадач, которой будет соответствовать аналогичная структура программных модулей (рис. 3.2).

Первый шаг детализации. Сначала наметим все необходимые подпрограммы, указав лишь их заголовки (спецификации). На ме-

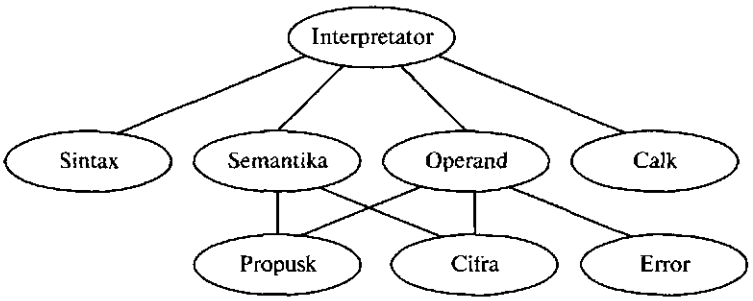


Рис. 3.2. Структура программных модулей

сте подпрограмм запишем поясняющие комментарии (такой вид подпрограммы называют заглушкой). Затем напишем основную часть программы.

```
Program Interpretator;
Type PozInt = 0..MaxInt;
Var Line : String;
  A, B : PozInt;
  Znak : Char;
  Flag : Boolean;
  Rezult : Integer;
Procedure Propusk(Stroka:String; M:Byte; Var
N:Byte);
Begin
{---Пропускает подряд стоящие пробелы в строке
Stroka, начиная с символа номер M. В переменной N
получает номер первого символа, не являющегося про-
белом----}
End;
Function Cifra(C : Char): Boolean;
Begin
{----Получает значение True, если символ C является
цифрой, и False – в противном случае----}
End;
Function Sintax(Stroka : String): Boolean;
Begin
{----Синтаксический контроль. Получает значение
True, если в строке нет других символов кроме цифр,
знаков операций, знака '=' и пробелов. В противном
случае – False----}
End;
Function Semantika(Stroka : String): Boolean;
Begin
{----Проверяет, соответствует ли структура стро-
ки формату 'a ⊕ b ='. Если соответствует, получает
значение True, если нет – False----}
End;
Procedure Operand(Stroka : String; Var A, B : PozInt;
   Var Z : Char; Var Flag : Boolean);
Begin
{----Выделяет операнды. Проверяет их на соответ-
ствие допустимому диапазону значений. Переменная
Flag получает значение True, если результат про-
```

```
верки положительный, и False - в противном случае.
Преобразует операнды в целые положительные числа A,
B. В переменной Z получает знак операции----}
End;
Procedure Calc(A, B : PozInt; Znak : Char; Var Rez:
  Integer; Var Flag : Boolean);
Begin
{----Вычисляет результат операции. Проверяет при-
надлежность результата диапазону от -MaxInt до
MaxInt. Flag - признак результата проверки. Резуль-
тат операции получается в переменной Rez----}
End;
Begin {----Начало основной части программы----}
  WriteLn('Введите выражение: ');
  WriteLn;
  Read(Line);
  If Not Sintax(Line)
  Then WriteLn('Недопустимые символы')
  Else If Not Semantika(Line)
  Then WriteLn('Неверное выражение')
  Else Begin
       Operand(Line, A, B, Znak, Flag);
       If Not Flag
       Then WriteLn('Слишком большие операнды')
       Else Begin
         Calc(A,B,Znak,Rezult,Flag);
         If Not Flag
         Then WriteLn('Большой результат')
         Else WriteLn(Rezult)
         End
       End
  End.
```

Второй шаг детализации — составление подпрограмм:

```
Procedure Propusk(Stroka:String; M:Byte; Var
N:Byte);
Begin
  N := M;
  While (Stroka[N] = ' ') And (N < Length(Stroka))
Do
   N :=  N + 1
End;
Function Cifra(C : Char): Boolean;
```

```
Begin
  Cifra := (C >= '0') And (C <= '9')
End;
Function Sintax(Stroka : String): Boolean;
Var I : Byte;
Begin
  Sintax := True;
  For I := 1 To Length(Stroka) Do
  If Not ((Stroka[I] = ' ') Or Cifra(Stroka[I]) Or
          (Stroka[I] = '+') Or (Stroka[I] = '-') Or
          (Stroka[I] = '*') Or (Stroka[I] = '='))
  Then Sintax := False
End;
Function Semantika(Stroka : String): Boolean;
Var I : Byte;
Begin
  Semantika := True;
  Propusk(Stroka, 1, I);
  If Not Cifra(Stroka[I])
  Then
       Begin
            Semantika := False;
            Exit
       End;
  While Cifra(Stroka[I]) Do
       I := I + 1;
  Propusk(Stroka, I, I);
  If Not((Stroka[I] = '+') Or (Stroka[I] = '-')
       Or (Stroka[I] = '*'))
  Then
  Begin
       Semantika := False;
       Exit
  End;
    I := I + 1;
    Propusk(Stroka, I, I);
    If Not Cifra(Stroka[I])
    Then
          Begin
               Semantika := False;
               Exit
          End;
    While Cifra(Stroka[I]) Do
```

```
            I := I + 1;
      Propusk(Stroka, I, I);
      If Stroka[I] <> '='
      Then
            Begin
                  Semantika := False;
                  Exit
            End;
      I := I + 1;
      Propusk(Stroka, I, I);
      If I < Length(Stroka)
      Then
            Begin
                  Semantika := False;
                  Exit
            End
End;
Procedure Operand(Stroka : String; Var A, B : PozInt;
Var Z: Char; Var Flag : Boolean);
Var P, Q : String;
     I : Byte;
     Code : Integer;
Function Error(S : String): Boolean;
{Функция принимает значение True, если в строке S
находится число, превышающее максимальное целое, и
False — в противном случае }
      Const Lim = '32767';
      Var I: Byte;
Begin
      If Length(S) > 5
      Then Error := True
      Else If Length(S) < 5
            Then Error := False
            Else Error := S > Lim
End;
Begin
      Flag := True;
      I := 1;
      Propusk(Stroka, I, I);
      Q := '';
      While Cifra(Stroka[I]) Do
            Begin
                  Q := Q + Stroka[I];
```

```
                I := I + 1
        End;
    If Error(Q)
    Then
        Begin
                Flag := False;
                Exit
        End;
    Propusk(Stroka, I, I);
    Z := Stroka[I];
    I := I + 1;
    Propusk(Stroka, I, I);
    P := '';
    While Cifra(Stroka[I]) Do
        Begin
                P := P + Stroka[I];
                I := I + 1
        End;
    If Error(P)
    Then
        Begin
                Flag := False;
                Exit
        End;
    Val(Q, A, Code);
    Val(P, B, Code)
{Стандартная процедура Val преобразует цифровую
строку в соответствующее число; Code - код ошибки}
End;
Procedure Calc(A, B : PozInt; Znak : Char;
    Var Rez : Integer; Var Flag : Boolean);
    Var X, Y, Z: Real;
Begin
    Flag := True;
    Y := A;
    Z := B;
    Case Znak Of
        '+': X := Y + Z;
        '-': X := Y - Z;
        '*': X := Y * Z
    End;
    If Abs(X) > MaxInt
    Then
```

```
            Begin
                  Flag := False;
                  Exit
            End;
      Rez := Round(X)
End;
```

Обратите внимание на то, что функция Error определена как внутренняя в процедуре Operand, так как она используется только в данной процедуре. Другим программным модулям она не требуется.

Окончательно объединив тексты подпрограмм с основной программой, получим рабочий вариант программы Interpretator. Теперь ее можно вводить в компьютер.

Отладка и тестирование программы. Никогда нельзя быть уверенным, что написанная программа сразу будет верной (хотя такое и возможно, но с усложнением задачи становится все менее вероятным). До окончательного рабочего состояния программа доводится в процессе *отладки*.

Ошибки в программе могут быть «языковые» и алгоритмические. Первые, как правило, помогает обнаружить компилятор с Паскаля. Это ошибки, связанные с нарушением правил языка программирования. Их также называют *ошибками времени компиляции*, так как обнаруживаются они именно во время компиляции. Сам компилятор в той или иной форме выдает пользователю сообщение о характере ошибки и ее месте в тексте программы. Исправив очередную ошибку, пользователь повторяет компиляцию. И так продолжается до тех пор, пока не будут ликвидированы все ошибки этого типа.

Алгоритмические ошибки могут приводить к различным последствиям. Например, могут возникать невыполнимые действия: деление на нуль, корень квадратный из отрицательного числа, выход индекса за границы строки и др. Это *ошибки времени исполнения*, и они приводят к прерыванию выполнения программы. Как правило, имеются системные программные средства, помогающие в поиске таких ошибок.

Также могут возникать ситуации, когда алгоритмические ошибки не приводят к прерыванию выполнения программы. Программа выполняется до конца, выдаются какие-то результаты, но они не верные. Для окончательной отладки алгоритма и анализа его правильности производится *тестирование*, т.е. выполняется *тест* — вариант решения задачи, для которого заранее известны результаты. Как правило, один тестовый вариант не может дока-

зать правильность программы. Следовательно, программист должен придумать систему тестов и соответственно план тестирования для исчерпывающего испытания всей программы.

Как уже говорилось, качественная программа ни в каком варианте не должна завершиться аварийно. Тесты программы Interpretator должны продемонстрировать, что при правильном вводе исходной строки всегда будут получаться правильные результаты, а при наличии ошибок (синтаксических, семантических, выхода за допустимый диапазон значений) — соответствующие сообщения. План тестирования нашей программы может быть, например, следующим:

№ п/п	Что вводим	Что должно получиться
1	25 + 73 =	98
2	5274 – 12315 =	– 7041
3	614 * 25 =	15350
4	25,5 + 31,2 =	Недопустимые символы
5	5 = 6 – 1	Неверное выражение
6	72315 – 256 =	Слишком большие операнды
7	32000 * 100 =	Большой результат

Правильное прохождение всех тестов — это необходимое условие правильности программы. Заметим, однако, что данное условие не всегда является достаточным. Чем сложнее программа, тем труднее построить исчерпывающий план тестирования. Опыт показывает, что даже в «фирменных» программах в процессе эксплуатации обнаруживаются ошибки, т.е. проблема тестирования программы — очень важная и одновременно очень сложная.

3.2. РЕКУРСИВНЫЕ МЕТОДЫ

Суть рекурсивных методов — это сведение задачи к самой себе.

Как уже говорилось, в Паскале существует возможность рекурсивного определения функций и процедур, реализующих рекурсивные алгоритмы. Однако создать рекурсивный алгоритм решения задачи часто бывает очень непросто.

Рассмотрим классическую задачу, известную в литературе под названием «Ханойская башня» (рис. 3.3).

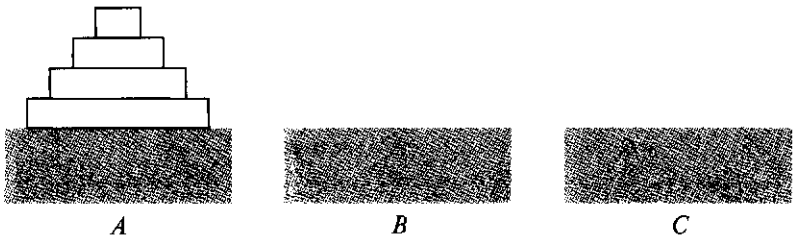


Рис. 3.3. Задача о ханойской башне

На площадке (назовем ее A) находится пирамида, составленная из дисков уменьшающегося от основания к вершине размера. Эту пирамиду в том же виде требуется переместить на площадку B. При выполнении этой работы необходимо соблюдать следующие ограничения:

- перекладывать можно только по одному диску, взятому сверху пирамиды;
- класть диск можно только или на основание площадки, или на диск большего размера;
- в качестве вспомогательной можно использовать площадку C.

Название «Ханойская башня» связано с легендой, согласно которой в давние времена монахи одного ханойского храма взялись переместить по этим правилам башню, состоящую из 64 дисков. Монахи все еще работают и еще долго будут работать!

Нетрудно решить данную задачу для двух дисков. Обозначив перемещения диска, например с площадки A на площадку B как $A \Rightarrow B$, запишем алгоритм для этого случая:

$$A \Rightarrow C;\ A \Rightarrow B;\ C \Rightarrow B,$$

т. е. всего три хода.

Алгоритм решения задачи для трех дисков будет длиннее:

$$A \Rightarrow B;\ A \Rightarrow C;\ B \Rightarrow C;\ A \Rightarrow B;\ C \Rightarrow A;\ C \Rightarrow B;\ A \Rightarrow B,$$

т. е. уже семь ходов.

Определить число ходов N для k дисков можно по следующей рекуррентной формуле:

$$N(1) = 1;\ N(\mathrm{k}) = 2 \times N(k - 1) + 1.$$

Так, получим число ходов $N(10) = 1023$, $N(20) = 104\,857$, а $N(64) = 1\,844\,6744\,073\,709\,551\,615$, т. е. столько перемещений придется сделать ханойским монахам. Попробуйте прочитать это число.

Теперь составим программу, по которой машина рассчитает алгоритм работы монахов и выведет его для любого числа дисков N. Пусть на площадке A находится N дисков, тогда алгоритм решения задачи будет следующим:

1. Если $N = 0$, то ничего не надо делать.
2. Если $N > 0$, то следует:

- переместить $N - 1$ дисков на C через B;
- переместить диск с A на B ($A \Rightarrow B$);
- переместить $N - 1$ дисков с C на B через A.

При выполнении второго пункта алгоритма последовательно будем иметь три этапа перемещения дисков, показанные на рис. 3.4.

Описание алгоритма имеет явно рекурсивный характер. Перемещение N дисков описывается через перемещение $N - 1$ дисков. А где же выход из этой последовательности рекурсивных ссылок алгоритма на самого себя? Он в первом пункте алгоритма, каким бы странным не казалось его тривиальное содержание.

А теперь построим программу решения рассматриваемой задачи на Паскале. В ней имеется рекурсивная процедура Hanoi, выполнение которой заканчивается только при $N = 0$. При обращении к этой процедуре используются фактические имена площадок, заданные соответственно их номерами: 1, 2, 3, поэтому на выходе цепочка перемещений будет описываться следующим образом:

$$1 \Rightarrow 2;\ 1 \Rightarrow 3;\ 2 \Rightarrow 3 \text{ и т.д.}$$

Программа решения задачи будет иметь следующий вид:

```
Program Monahi;
Var N: Byte;
Procedure Hanoi(N: Byte; A, B, C: Char);
Begin
  If N > 0 Then
  Begin Hanoi(N-1, A, C, B);
        WriteLn(A, '=>', B);
        Hanoi(N-1, C, B, A)
  End
End;
Begin
  WriteLn('Укажите число дисков: ');
  ReadLn(N);
  Hanoi(N, '1', '2', '3')
End.
```

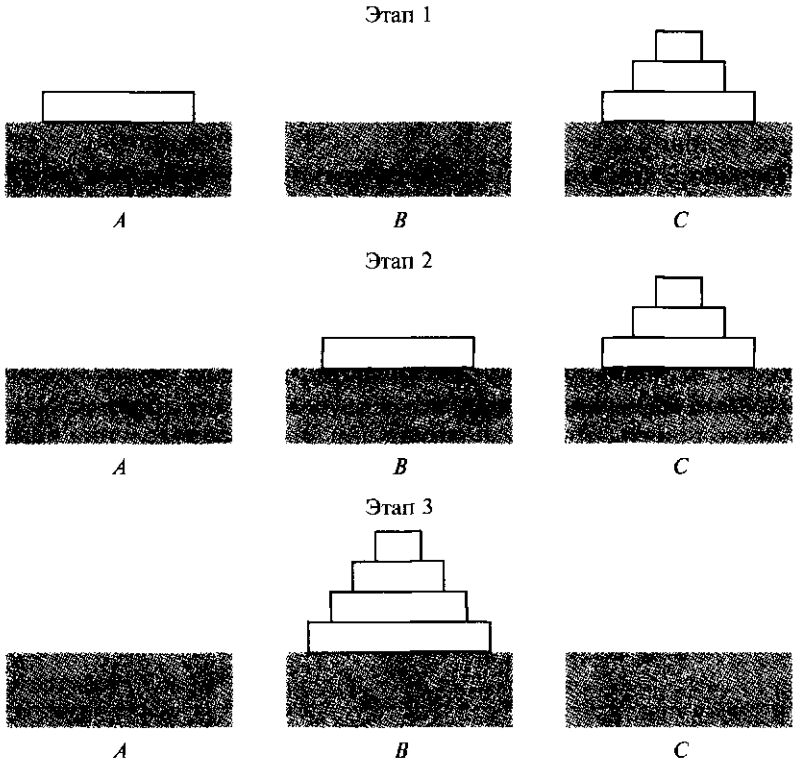


Рис. 3.4. Три этапа перемещения дисков

Это одна из самых удивительных программ! Попробуйте воспроизвести ее на машине. Проследите, как изменяется число ходов с ростом N, для чего можете сами добавить в программу счетчик ходов и в конце вывести его значение или печатать ходы с порядковыми номерами.

3.3. МЕТОДЫ ПЕРЕБОРА В ЗАДАЧАХ ПОИСКА

Рассмотрим некоторые задачи, связанные с проблемой поиска информации. Это огромный класс задач, достаточно подробно описанный в классической литературе по программированию.

Общий смысл задач поиска сводится к следующему: из данной информации, хранящейся в памяти ЭВМ, выбрать необходимые сведения, удовлетворяющие определенным условиям (критериям).

Подобные задачи мы уже рассматривали. Например, поиск максимального числа в числовом массиве, поиск необходимой записи в файле данных и т.д. Такой поиск осуществляется перебором всех элементов структуры данных и их проверкой на соблюдение условий поиска.

Перебор, при котором просматриваются все элементы структуры и который называется *полным перебором*, является «лобовым» способом поиска и, очевидно, не всегда самым лучшим.

Рассмотрим пример. В одномерном массиве *X* заданы координаты *N* точек, лежащих на вещественной числовой оси. Точки пронумерованы. Их номера соответствуют последовательности в массиве *X*. Требуется определить номер первой точки, наиболее удаленной от начала координат.

Легко понять, что это знакомая нам задача определения номера наибольшего по модулю элемента массива *X*, которая решается посредством полного перебора:

```
Const N = 100;
Var X : Array[1..N] Of Real; I, Number : 1..N;
Begin
{----Ввод массива X----}
.....................
Number := 1;
For I := 2 To N Do
    If Abs(X[I]) > Abs(X[Number])
    Then Number := I;
    WriteLn(Number)
End.
```

Здесь полный перебор элементов одномерного массива производится с помощью одной циклической структуры.

А теперь при тех же исходных данных определим пару точек, расстояние между которыми наибольшее.

Применяя метод перебора, эту задачу можно решить следующим образом: перебрать все пары точек из *N* заданных и определить номера тех из них, расстояние между которыми наибольшее (наибольший модуль разности координат). Такой полный перебор реализуется через два вложенных цикла:

```
Number1 := 1;
Number2 := 2;
For I := 1 To N Do
  For J := 1 To N Do
```

```
If Abs(X[I] - X[J] > Abs(X[Number1] - X[Number2])
Then
Begin
   Number1 := I;
   Number2 := J
End;
```

Однако очевидно, что такое решение задачи нерационально. Здесь каждая пара точек будет просматриваться дважды, например при $I = 1$, $J = 2$ и $I = 2$, $J = 1$. Для случая $N = 100$ выполнение циклов будет повторяться $100 \times 100 = 10\,000$ раз.

Выполнение программы ускорится, если исключить повторения. Исключить также следует и совпадения значений I и J. Тогда число повторений цикла будет равно $\frac{N(N-1)}{2}$, т.е. при $N = 100$ получится 4 950 повторений.

Для исключения повторений в предыдущей программе следует изменить начало внутреннего цикла с 1 на $I + 1$. Тогда программа примет следующий вид:

```
Number1 := 1;
Number2 := 2;
For I := 1 To N Do
  For J := I + 1 To N Do
    If Abs(X[I] - X[J]) > Abs(X[Number1] -
X[Number2])
    Then
    Begin
      Number1 := I;
      Number2 := J
    End;
```

Рассмотренный вариант алгоритма назовем *перебором без повторений*.

Примечание. Конечно, эту задачу можно было решить и другим способом, но в данном случае нас интересовал именно переборный алгоритм. Для точек, расположенных не на прямой, а на плоскости или в пространстве, поиск альтернативы переборному алгоритму становится весьма проблематичным.

В следующей задаче требуется выбрать все тройки чисел без повторений из массива X, сумма которых равна десяти.

В этом случае алгоритм будет состоять из трех вложенных циклов, имеющих переменную длину:

```
For I := 1 To N Do
  For J := I + 1 To N Do
      For K := J + 1 To N Do
          If X[I] + X[J] + X[K] = 10
          Then WriteLn(X[I], X[J], X[K]);
```

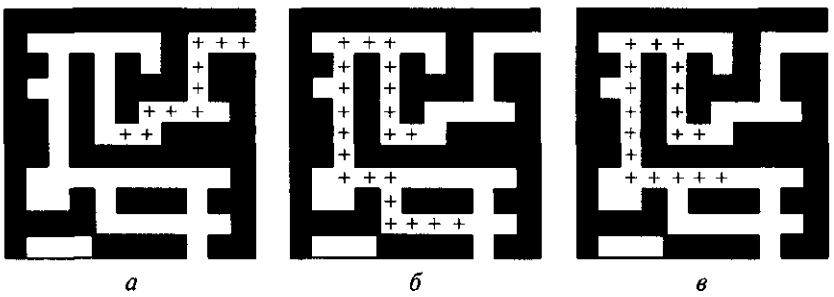
А теперь представьте, что из массива X требуется выбрать все группы чисел, сумма которых равна десяти. Количество чисел в группах может быть любым, т.е. от 1 до N. В этом случае количество вариантов перебора резко возрастает, а сам алгоритм становится нетривиальным.

Казалось бы, ну и что? Машина работает быстро! И все же посчитаем. Число различных групп из N объектов (включая пустую) составляет 2^N. При $N = 100$ это будет $2^{100} \approx 10^{30}$. Следовательно, компьютер, работающий со скоростью миллиард операций в секунду, будет осуществлять такой перебор приблизительно 10 лет. Даже исключение перестановочных повторений не сделает такой переборный алгоритм практически осуществимым.

Путь практической разрешимости подобных задач состоит в нахождении способов исключения из перебора бесперспективных с точки зрения условия задачи вариантов. Для некоторых задач это удается сделать с помощью алгоритма, описанного далее.

Рассмотрим алгоритм *перебора с возвратом* на примере задачи о прохождении лабиринта (рис. 3.5).

Дан лабиринт, оказавшись внутри которого необходимо найти выход из него. Перемещаться можно только в горизонтальном и вертикальном направлениях. Все варианты путей выхода из центральной точки лабиринта показаны на рис. 3.5, *а*...*в*.



а *б* *в*

Рис. 3.5. Варианты (*а*...*в*) путей прохождения лабиринта

При создании программы решения этой задачи встают два вопроса: как организовать данные и как построить алгоритм?

Информацию о форме лабиринта будем хранить в квадратной матрице LAB символьного типа, размером $N \times N$, где N — нечетное число (чтобы иметь центральную точку). На профиль лабиринта накладывается сетка таким образом, чтобы в каждой ее ячейке находилась либо стена, либо проход. Матрица отражает заполнение сетки: элементами прохода соответствуют пробел, а элементам стены, — какой-нибудь символ, например буква «М». Путь движения по лабиринту будет отмечаться символами «+». Например, рис. 3.5, *б* соответствует заполнению матрицы LAB, приведенному далее.

М	М	М	М	М	М	М	М	М	М	М
М		+	+	+			М			
М	М	+	М	+	М		М		М	М
М		+	М	+	М	М	М		М	М
М	М	+	М	+	М					М
М	М	+	М	+	+		М	М	М	М
М	М	+	М	М	М	М	М	М	М	М
М	М	+	+	+						М
М			М	+	М	М	М		М	М
М	М	М	М	+	+	+	+	+		М
М				М	М	М	М	+	М	М

Исходные данные для анализа — профиль лабиринта (исходная матрица LAB без крестиков); требуемый результат — все возможные траектории выхода из центральной точки лабиринта (для каждого пути выводится матрица LAB с траекторией, отмеченной крестиками).

Алгоритм перебора с возвратом также называют *методом проб*, и суть его состоит в следующем.

1. Из каждой очередной точки траектории просматриваются возможные направления движения в одной и той же последовательности. Договоримся, что просмотр будет происходить каждый раз против часовой стрелки (справа — сверху — слева — снизу); шаг производится в первую же обнаруженную свободную соседнюю клетку; клетка, в которую сделан шаг, отмечается крестиком.

2. Если из очередной клетки дальше пути нет (тупик), следует возврат на один шаг назад и просмотр еще не испробованных путей движения из этой точки. При возвращении назад покинутая клетка отмечается пробелом.

3. Если очередная клетка, в которую сделан шаг, оказалась на краю лабиринта (на выходе), на печать выводится найденный путь.

Программу будем строить методом последовательной детализации. Первый этап детализации:

```
Program Labirint;
Const N = 11; {размер лабиринта N × N клеток}
Type Field = Array[1..N, 1..N] Of Char;
Var LAB : Field;
Procedure GO(LAB : Field; X, Y : Integer);
Begin
{Поиск путей из центра лабиринта до края – каждый
найденный путь печатается}
End;
Begin
  {Ввод лабиринта}
  GO(LAB, N Div 2 + 1, N Div 2 + 1)  {Начинаем с
                                      середины}
End.
```

Процедура GO пытается сделать шаг в клетку с координатами (*X*, *Y*). Если эта клетка оказывается на выходе из лабиринта, выводится на печать пройденный путь, если же нет, то делается шаг в соседнюю клетку в соответствии с установленной ранее последовательностью. Если клетка тупиковая, делается шаг назад. Из всего сказанного следует, что процедура носит рекурсивный характер.

Запишем сначала общую схему процедуры без детализации:

```
Procedure GO(LAB : Field; X, Y : Integer);
Begin
  If {Клетка (X, Y) свободна}
  Then
  Begin
       {Шаг на клетку (X, Y)}
       If {Дошли до края лабиринта}
       Then {Печатается найденный путь }
       Else {Попытка сделать шаг в соседние клетки в
```

```
            условленной последовательности}
            {Возвращение на один шаг назад}
    End
End;
```

Для вывода найденных траекторий движения составляется процедура PRINTLAB.

В окончательном виде программа будет иметь следующий вид:

```
Program Labirint;
Const N = 11; {Размер лабиринта N ×N клеток}
Type Field = Array[1..N, 1..N] Of Char;
Var LAB : Field; X, Y : Integer;
Procedure PRINTLAB(LAB : Field);
{Печать найденного пути в лабиринте}
Var X, Y : Integer;
Begin
 For X := 1 To N Do
 Begin
   For Y := 1 To N Do
       Write(LAB[X, Y]);
       WriteLn
 End;
 WriteLn
End; {Печати}
Procedure GO(LAB : Field; X, Y : Integer);
Begin
 If LAB[X, Y] = ' ' {Если клетка свободна}
 Then
 Begin
   LAB[X, Y] := '+'; {Делается шаг}
   If (X = 1) Or (X = N) Or (Y = 1) Or (Y = N) {Край}
   Then
       PRINTLAB(LAB){Печатается найденный путь}
   Else
       Begin {Поиск следующего шага}
             GO(LAB, X + 1, Y);
             GO(LAB, X, Y + 1);
             GO(LAB, X-1,Y);
             GO(LAB, X, Y-1)
       End;
   LAB[X, Y] := ' ' {Возвращение назад}
 End
```

```
End; {Процедуры GO}
Begin  {Основной программы}
       {Ввод лабиринта}
 For X := 1 To N Do
 Begin
   For Y := 1 To N Do
       Read(LAB[X, Y]);
       ReadLn
 End;
 GO(LAB,N Div 2 + 1, N Div 2 + 1){Начинаем с сере-
дины}
End.
```

Это еще один пример красивой программы с использованием рекурсивного определения процедуры.

Схема алгоритма данной программы типична для метода перебора с возвратом. По аналогичным алгоритмам решаются, например, популярные задачи об обходе шахматной доски фигурами или о расстановке фигур на доске таким образом, чтобы они «не били» друг друга, а также множество задач оптимального выбора (задачи о коммивояжере, об оптимальном строительстве дорог и т.д.).

Примечание. Из-за использования массива LAB в качестве параметра-значения в процедуре GO могут возникнуть проблемы с памятью при исполнении программы на ЭВМ. В этом случае следует перейти к глобальной передаче массива.

3.4. МЕТОДЫ СОРТИРОВКИ ДАННЫХ И СЛОЖНОСТЬ АЛГОРИТМОВ

Сложность алгоритмов. Сначала обсудим проблему оценки сложности алгоритма. Традиционно принято оценивать степень сложности алгоритма по объему используемых им основных ресурсов компьютера: процессорного времени и оперативной памяти. В связи с этим вводятся понятия *временно́й* и *объемной сложности* алгоритма.

Временна́я сложность становится особенно важна для задач, предусматривающих интерактивный режим работы программы, или для задач управления в режиме реального времени. Часто программисту, составляющему программу управления каким-нибудь техническим устройством, приходится искать компромисс между точностью вычислений и временем работы программы. Как правило, повышение точности ведет к увеличению времени.

Объемная сложность программы становится критической, когда объем обрабатываемых данных оказывается на пределе объема оперативной памяти ЭВМ. Для современных компьютеров острота этой проблемы снижается благодаря росту объема их ОЗУ и эффективному использованию многоуровневой системы запоминающих устройств. При этом программе становится доступной очень большая, практически неограниченная область памяти (виртуальная память), и недостаток основной памяти приводит лишь к некоторому замедлению работы из-за выполнения обменов с диском. Существуют приемы, позволяющие минимизировать потери времени при таком обмене: использование кеш-памяти и аппаратного просмотра команд программы вперед, что позволяет заблаговременно переносить с диска в основную память необходимые значения. Из сказанного можно заключить, что минимизация емкостной сложности не является первоочередной задачей, поэтому далее мы будем интересоваться в основном временно́й сложностью алгоритмов.

Время выполнения программы пропорционально числу исполняемых операций. Разумеется, в единицах времени (с) оно также зависит и от скорости работы процессора. Для того чтобы показатель временно́й сложности алгоритма был инвариантен техническим характеристикам компьютера, его измеряют в относительных единицах. Обычно временна́я сложность оценивается числом выполняемых операций.

Как правило, временна́я сложность алгоритма зависит от исходных данных, причем как от величин, так и от их объема. Если обозначить значение параметра временнЬй сложности алгоритма α символом T_α, а некоторый числовой параметр, характеризующий исходные данные, буквой V, временну́ю сложность можно представить как функцию $T_\alpha(V)$. Выбор параметра V зависит от типа решаемой задачи или вида используемого для решения алгоритма.

Пример 3.1. Требуется оценить временну́ю сложность алгоритма вычисления факториала целого положительного числа.

Программа решения:

```
Function Factorial(x : Integer) : Integer;
Var m, i : Integer;
Begin
  m := 1;
  For i := 2 To x Do m := m * i;
  Factorial := m
End;
```

Подсчитаем общее число операций, выполняемых программой при данном значении x. Один раз выполняется оператор $m := 1$; тело цикла (в котором две операции: умножение и присваивание) выполняется $(x - 1)$ раз; один раз выполняется присваивание Factorial := m. Если каждую из операций принять за единицу сложности, временна́я сложность всего алгоритма будет иметь вид $1 + 2(x - 1) + 1 = 2x$, откуда понятно, что в качестве параметра V следует принять значение x. Функция временно́й сложности получится следующей:

$$T_\alpha(V) = 2V.$$

В этом случае можно сказать, что временна́я сложность линейно зависит от параметра данных — значения аргумента функции Factorial.

Пример 3.2. Требуется вычислить скалярное произведение двух векторов: $A = (a_1, a_2, ..., a_k)$, $B = (b_1, b_2, ..., b_k)$.

Программа решения:

```
AB := 0;
For i := 1 To k Do AB := AB + A[i] * B[i];
```

В этой задаче объем входных данных $n = 2k$. Число выполняемых операций $1 + 3k = 1 + 3(n/2)$. Здесь можно принять $V = k = n/2$. Сложность алгоритма не зависит от значений элементов векторов A и B. Как и в примере 3.1, здесь можно говорить о линейной зависимости временно́й сложности от параметра данных.

С параметром временно́й сложности алгоритма обычно связывают две теоретические проблемы. Первая проблема состоит в поиске ответа на следующий вопрос: до какого предела значения временно́й сложности можно дойти, совершенствуя алгоритм решения задачи? Этот предел зависит от самой задачи и, следовательно, является ее собственной характеристикой.

Вторая проблема связана с классификацией алгоритмов по временно́й сложности. Функция $T_\alpha(V)$ обычно растет с ростом V. Как быстро она растет? Существуют алгоритмы с линейной зависимостью T_α от V (как в рассмотренных примерах), с квадратичной зависимостью и зависимостью более высоких степеней, которые называются полиномиальными. Существуют также алгоритмы, сложность которых растет быстрее любого полинома. Проблема, которую часто решают исследователи алгоритмов, заключается в следующем вопросе: возможен ли для таких задач полиномиальный алгоритм?

Постановка задачи сортировки данных. Существует традиционное деление алгоритмов на две категории: численные и нечисленные. *Численные алгоритмы* предназначены для математических расчетов: вычисления по формулам, решения уравнений,

статистической обработки данных и т.п. В таких алгоритмах основным типом обрабатываемых данных являются числа. *Нечисленные алгоритмы* имеют дело с самыми разнообразными типами данных: символьными, графическими, мультимедийной информацией. К этой категории относятся многие алгоритмы системного программирования (трансляторы, операционные системы), системы управления базами данных, сетевого программного обеспечения и др.

Для программных продуктов второй категории наиболее часто используются алгоритмы сортировки данных, т.е. упорядочения информации по некоторому признаку. От эффективности этих алгоритмов, и прежде всего от скорости их выполнения, во многом зависит эффективность работы всей программы.

Различают алгоритмы внутренней сортировки — сортировки во внутренней памяти — и алгоритмы внешней сортировки — сортировки файлов. Далее будем рассматривать только внутреннюю сортировку.

Как правило, сортируемые данные располагаются в массивах. В простейшем случае — это числовые массивы. Однако для нечисленных алгоритмов более характерна сортировка массива. Поле, по значению которого производится сортировка, называется ключом сортировки и обычно имеет числовой тип. Например, массив сортируемых записей содержит два поля: наименование товара и количество товара на складе. В программе на Паскале его описание имеет следующий вид:

```
Const n = 1000;
Type Tovar = Record
                Name : String;
                Key : Integer
             end;
Var A : Array[1..n] Of Tovar;
```

Сортировка данных производится либо по возрастанию, либо по убыванию значения ключа A[i].key.

Далее во всех примерах программ предполагается, что приведенные ранее описания в программе присутствуют глобально и их область действия распространяется на процедуры сортировки.

Алгоритм сортировки методом «пузырька» рассматривался в подразд. 2.16. Теперь обсудим сортировку простым включением и быструю сортировку.

Алгоритм сортировки простым включением. Предположим, что на некотором этапе работы алгоритма левая часть массива с

1-го по (i – 1)-й элемент отсортирована, а правая его часть с i-го по n-й элемент остается неотсортированной. Очередной шаг алгоритма заключается в расширении левой части массива на один элемент и соответственно сокращении правой части. Для этого берется первый элемент правой части (с индексом i) и вставляется на подходящее место в левую часть таким образом, чтобы ее упорядоченность сохранилась. Процесс начинается с левой части массива, состоящей из одного элемента $A[1]$, а заканчивается, когда его правая часть становится пустой.

Программа сортировки простым включением следующая:

```
Procedure StraightInsertion;
Var i, j : Integer; x : Tovar;
Begin     For i := 2 To n Do
  Begin x := A[i]; {В переменной x запоминается
значение, которое необходимо поставить на свое ме-
сто в левой части}
     j := i - 1; {Правый край левой части}
     While (x.key < A[j].key) and (j >= 1) Do
     Begin A[j + 1] := A[j]; j := j - 1; {Продвиже-
ние «дырки» в левой части массива справа налево до
той позиции, в которую должен быть включен элемент
A[i]}
     End;
     A[j + 1] := x {Включение A[i] в «дырку» в ле-
вой части}
  End
End;
```

Теперь оценим сложность алгоритма сортировки простым включением. Очевидно, что временна́я сложность зависит как от размера сортируемого массива, так и от его исходного состояния, т.е. упорядоченности элементов. Временна́я сложность будет минимальной, если исходный массив уже отсортирован в требуетмом порядке значений ключа (в данном случае по возрастанию). Максимальное значение временно́й сложности соответствует противоположной упорядоченности исходного массива, т.е. в этом случае оно максимально, если исходный массив упорядочен по убыванию значений ключа. Обычно временна́я сложность алгоритмов сортировки оценивается числом пересылок элементов.

Оценим минимальную временну́ю сложность алгоритма. Если массив уже отсортирован, тело цикла **While** не будет выполняться

ни разу. Выполнение процедуры сведется к работе следующего цикла:

```
For i := 2 To n Do
Begin
  x := A[i];
  j := i-1;
  A[j + 1] := x
End;
```

Поскольку тело цикла **For** исполняется $n - 1$ раз, то число пересылок элементов массива $M_{min} = 2(n - 1)$, а число сравнений ключей $C_{min} = n - 1$.

Сложность алгоритма будет максимальной, если исходный массив упорядочен по убыванию. Тогда каждый элемент $A[i]$ будет «прогоняться» к началу массива, т. е. устанавливаться в первую позицию. Цикл **While** выполнится один раз при $i = 2$, два раза при $i = 3$ и $n - 1$ раз при $i = n$. Таким образом, общее число пересылок записей

$$M_{max} = 2(n-1) + \sum_{i=2}^{n}(i-1) = 2(n-1) + n(n-1)/2 = \frac{1}{2}(n^2 + 3n - 4).$$

Более приемлемой в реальной ситуации является *средняя оценка сложности алгоритма*. Для ее вычисления следует предположить, что все элементы исходного массива — случайные числа и их значения никак не связаны с их номерами. В этом случае результат очередной проверки условия x.key < A[j].key в цикле **While** также будет являться случайным.

Логично допустить, что среднее число выполнений цикла **While** для каждого конкретного значения i равно $i/2$, т. е. в среднем каждый раз приходится просматривать половину массива до тех пор, пока не найдется подходящее место для очередного элемента. Тогда формула для вычисления среднего числа пересылок (средняя оценка сложности) будет следующей:

$$M_{ср} = 2(n-1) + \sum_{i=2}^{n}\frac{i}{2} = 2(n-1) + (n-2)(n-1)/4 = \frac{1}{4}(n^2 + 9n - 10).$$

Максимальная и средняя оценки сложности алгоритма квадратичны (полином второй степени) по параметру n — размеру сортируемого массива.

Алгоритм быстрой сортировки. Этот алгоритм был разработан Э. Хоаром. В алгоритме быстрой сортировки используются:

- разделение сортируемого массива на две части — левую и правую;
- взаимное упорядочение двух частей (подмассивов) таким образом, чтобы все элементы левой части не превосходили элементов правой части;
- рекурсия, при которой подмассив упорядочивается точно таким же способом, как и весь массив.

Для разделения массива на две части необходимо выбрать некоторое барьерное значение ключа. Это значение должно удовлетворять единственному условию: находиться в диапазоне значений для данного массива (т.е. между минимальным и максимальным значениями). В качестве барьерного можно выбрать значение ключа любого элемента массива, например первого, последнего или находящегося в его середине.

Далее следует сделать так, чтобы в левом подмассиве оказались все элементы с ключом, меньше барьера, а в правом — с ключом, больше «барьера». Затем, просматривая массив слева направо, надо найти позицию первого элемента с ключом, больше «барьера», а просматривая его справа налево, найти первый элемент с ключом, меньше «барьера». Поменяв эти значения местами, продолжить встречное движение до следующей пары элементов, предназначенных для обмена. Процесс необходимо продолжать пока индексы левого и правого просмотров не совпадут. Место их совпадения станет границей между двумя взаимно упорядоченными подмассивами. Далее алгоритм рекурсивно применяется к каждому из подмассивов (левому и правому). В итоге получим совокупность из n взаимно упорядоченных одноэлементных массивов, которые делить дальше невозможно. Эта совокупность образует один полностью упорядоченный массив, т.е. сортировка завершена.

Программа быстрой сортировки данных имеет следующий вид:

```
Procedure Quicksort;
Procedure Sort(L, R : Integer);
Var i, j, bar : Integer; w : Tovat;
Begin
  i := L;
  j := R;
  bar := A[(L+R)div2].key; {Установка барьера}
  Repeat
    {Поиск элемента слева для обмена}
```

```
      While A[i].key < bar Do i := i + 1;
      {Поиск элемента справа для обмен}
      While A[j].key > bar Do j := j - 1;
      {Обмен элементов и смещение по массиву}
      If i <= j Then
      Begin
        w := A[i]; A[i] := A[j]; A[j] := w;
        i := i + 1; j := j - 1
      End;
    Until i > j;
    {Сформированы взаимно упорядоченные подмассивы}
    {Сортировка левого подмассива}
      If L < j Then Sort(L,j);
        {Сортировка правого подмассива}
        If i < R Then Sort(i,R);
  End; {Sort}
  Begin
    Sort(1,n)
  End; {Quicksort}
```

Исследование временно́й сложности алгоритма быстрой сортировки является очень трудоемкой задачей, поэтому здесь мы его приводить не будем. Дадим лишь окончательный результат этого анализа. Временна́я сложность T как функция от n — размера массива — по порядку величины выражается следующей формулой:

$$T(n) = O(n\ln(n)).$$

Здесь использовано принятое в математике обозначение $O(x)$ — величина порядка x. Следовательно, временна́я сложность алгоритма быстрой сортировки есть величина порядка $n\ln(n)$, что для целых положительных чисел n меньше, чем n^2 (вспомним, что алгоритм сортировки простым включением имеет сложность порядка n^2). И чем больше значение n, тем эта разница существеннее. Например:

$n = 10,$	$n^2 = 100,$	$n\ln(n) = 23{,}03;$
$n = 100,$	$n^2 = 10\,000,$	$n\ln(n) = 460{,}52.$

УПРАЖНЕНИЯ

1. Даны декартовы координаты N точек на плоскости. Составить программы решения следующих задач:

а) найти две самые близкие друг к другу точки;

б) найти две самые удаленные друг от друга точки;

в) найти три точки, лежащие в вершинах треугольника с наибольшим периметром;

г) найти две ближайшие точки, отрезок между которыми может служить радиусом окружности, заключающей внутри себя все остальные точки, и указать, какая из них является центральной.

2. Изменить программу Labirint таким образом, чтобы на печать выводился лишь кратчайший путь из центра лабиринта до края.

3. Составить программу, в соответствии с которой шахматный конь обойдет всю доску, побывав на каждом поле всего один раз.

4. Составить программу расстановки на шахматной доске восьми ферзей таким образом, чтобы они не угрожали друг другу.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем состоит сущность метода последовательной детализации в программировании?
2. Каким образом производится отладка программы?
3. В чем заключается задача о Ханойской башне? Сформулируйте идею применения рекурсии для программирования задачи о Ханойской башне.
4. Что такое задача полного перебора?
5. В чем состоит идея алгоритма перебора с возвратом? Поясните на примере задачи о прохождении лабиринта.
6. Какие существуют критерии сложности алгоритмов?
7. Что такое алгоритм с полиномиальной временной сложностью?
8. Какова идея алгоритма сортировки простым включением?
9. В чем заключается идея алгоритма быстрой сортировки?
10. Каков порядок сложности алгоритма быстрой сортировки?

Глава 4

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Из данной главы вы узнаете:

- что такое объектно-ориентированное программирование;
- историю объективно-ориентированного программирования;
- основные понятия объектно-ориентированного программирования — объект, состояние объекта, методы объекта, инкапсуляция, класс, наследование, полиморфизм;
- правила описания и использования объектов в языке Object Pascal;
- историю и назначение Delphi интегрированной среды программирования;
- об основных элементах интерфейса Delphi;
- об основных компонентах Delphi и их свойствах;
- что такое событийно-управляемое программирование и как оно реализуется в Delphi;
- об этапах разработки программного приложения в Delphi;
- об иерархии классов.

Вы научитесь:

- описывать и использовать объекты в несложных программах на Object Pascal;
- работать в среде программирования Delphi;
- создавать несложный интерфейс для приложений Delphi;
- определять свойства и программировать методы для объектов интерфейса;
- создавать несложные приложения Windows средствами Delphi.

4.1. ЧТО ТАКОЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

История ООП. Основоположниками объектного подхода в программировании считаются норвежцы Оле Джохан Дал и Кристен Нюгорт — авторы языка Симула. История Симулы началась в 1962 г. с проекта Simulation Language, предназначенного для программного моделирования метода Монте-Карло. В 1965 г. у авторов этого проекта возникла идея объединить данные с процедурами, их обрабатывающими. В язык были введены новые средства моделирования и имитации мультипроцессорной работы, а также термины «класс» и «объект». Тогда же возникла и технология наследования, т.е. создатели Симулы ввели в язык возможность использования разными классами общих свойств посредством указания названия класса в виде префикса.

Алан Кей ввел новую концепцию разработки программ, в соответствии с которой набор последовательно выполняющихся инструкций мог быть заменен на многомерную среду взаимодействия объектов, общающихся друг с другом посредством асинхронного обмена сообщениями. Эту идею он воплотил в новом языке SmallTalk, первоначально названном им Biological System и смоделированном на Бейсике, а затем реализованном на ассемблере. Термин «объектно-ориентированное программирование» впервые был применен именно по отношению к языку SmallTalk. Наиболее известна версия языка SmallTalk 80.

В 1980 г. Бьерн Страуструп дополнил язык процедурного программирования Си концепцией классов, а в 1983 г. дал этому новому продукту окончательное название Си++. С тех пор Си++ стал наиболее популярным языком объектно-ориентированного программирования.

В дальнейшем возможности объектного программирования стали включаться и в другие языки. В Турбо Паскале средства ООП появились с версии 5.5. Элементы ООП включены в Фортран, начиная с версии Фортран-90. Современными языками программирования, объединившими технологию ООП с визуальными методами программирования, являются Delphi, Visual Basic, Java.

Основные понятия ООП. Основополагающей идеей объектно-ориентированной парадигмы программирования является объединение данных и обрабатывающих их процедур в единое целое — *объекты*.

Объектно-ориентированное программирование — это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса (типа данных особого вида), а классы образуют иерархию, основанную на принципах наследования. При этом объект характеризуется как совокупностью всех своих свойств и их текущих значений, так и совокупностью допустимых для данного объекта действий.

Несмотря на то что в различных литературных источниках делается акцент на те или иные особенности внедрения и применения ООП, три основных (базовых) его понятия остаются неизменными:

- *инкапсуляция* (Encapsulation);
- наследование (Inheritance);
- полиморфизм (Polymorphism).

Эти понятия лежат в основе ООП.

При процедурном подходе к программированию требуется описать каждый шаг, каждое действие алгоритма для достижения конечного результата. При объектно-ориентированном подходе право решать, как отреагировать и что сделать в ответ на поступивший вызов, остается за объектом. Для этого достаточно в стандартной форме поставить перед ним задачу и получить ответ.

Объект состоит из трех частей:

- имени;
- *состояния (переменных состояния)*;
- методов (операций).

Можно дать обобщающее определение: *объект ООП* — это совокупность переменных состояния и связанных с ними методов (операций), которые определяют, как объект взаимодействует с окружающим миром.

Методы объекта — это процедуры и функции, объявление которых включено в описание объекта и которые выполняют действия. Возможность управлять состояниями объекта посредством вызова методов в итоге и определяет поведение объекта. Эту совокупность методов часто называют *интерфейсом объекта*.

Инкапсуляция — это механизм, который объединяет данные и методы, манипулирующие этими данными, и защищает и то и другое от внешнего вмешательства или неправильного использования. Когда методы и данные объединяются таким способом, создается объект.

Применяя инкапсуляцию, мы защищаем данные, принадлежащие объекту, от возможных ошибок, которые могут возникнуть при прямом доступе к этим данным. Кроме того, применение этого механизма очень часто помогает локализовать возможные ошибки в коде программы, что намного упрощает процесс их поиска и исправления. Можно сказать, что под инкапсуляцией подразумевается сокрытие данных, т.е. их защита. Применение инкапсуляции ведет к снижению эффективности доступа к элементам объекта, что обусловлено необходимостью вызова методов для изменения его внутренних элементов (переменных). Однако при современном уровне развития вычислительной техники эти потери в эффективности не играют существенной роли.

Наследование — это процесс, посредством которого один объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него. В итоге создается иерархия объектных типов, где поля данных и методов «предков» автоматически являются и полями данных и методов «потомков».

Смысл и универсальность наследования заключается в том, что не надо каждый раз заново (с нуля) описывать новый объект, а можно указать «родителя» (базовый класс) и описать отличительные особенности нового класса. В результате новый объект будет обладать всеми свойствами «родительского» класса и своими собственными отличительными особенностями.

Например, можно создать базовый класс «Транспортное средство», который будет универсальным для всех средств передвижения на четырех колесах. Этот класс «знает», как движутся колеса, как они поворачивают, тормозят и т.д. Затем на основе этого класса можно создать класс «Легковой автомобиль». Поскольку новый класс унаследован от класса «Транспортное средство», то унаследованы и все основные черты этого класса, т.е. не надо в очередной раз описывать, как движутся колеса и т.д., а следует просто добавить те черты, особенности поведения, которые характерны для легковых автомобилей. В то же время можно, взяв за основу тот же класс «Транспортное средство», построить класс «Грузовые автомобили», описав их отличительные особенности, и на основании уже этого нового класса описать определенный подкласс грузовиков и т.д. Таким образом, сначала формируется простой шаблон, а затем посредством усложнений и конкретизации поэтапно создаются все более сложные шаблоны.

Полиморфизм — это свойство, которое позволяет одно и то же имя использовать для решения нескольких технически разных задач. Полиморфизм подразумевает такое определение ме-

тодов в иерархии типов, при котором метод с одним именем может применяться к различным родственным объектам. В общем виде концепцией полиморфизма является идея «один интерфейс — множество методов», т.е. полиморфизм помогает снижать сложность программ, разрешая использование одного интерфейса для единого класса действий. При этом выбор конкретного действия в зависимости от ситуации возлагается на компилятор.

Например, пусть есть класс «Автомобиль», в котором описано, как должен передвигаться автомобиль, как он поворачивает, как подает сигнал и т.д. Там же описан метод «Переключение передач». Допустим, что в этом методе класса «Автомобиль» описана автоматическая коробка передач. Требуется описать класс «Спортивный автомобиль», у которого механическое (ручное) переключение скоростей. Конечно, можно было бы описать заново все методы для нового класса. Однако вместо этого можно указать, что класс «Спортивный автомобиль» унаследован от класса «Автомобиль», а следовательно, обладает всеми свойствами и методами, описанными для «родительского» класса. Единственное, что надо сделать — это переписать метод «Переключение передач» для механической коробки передач. В результате при вызове метода «Переключение передач» будет выполняться метод не родительского класса, а вновь созданного.

Механизм работы ООП в этих случаях можно описать примерно таким образом. При вызове того или иного метода класса сначала следует искать метод в самом классе. Если метод найден, он выполняется и поиск его завершается. Если же метод не найден, следует обратиться к «родительскому» классу и искать вызванный метод в нем. Если метод найден, он выполняется и поиск его также прекращается, а если не найден, поиск продолжается далее вверх по иерархическому дереву, вплоть до корня (верхнего класса) иерархии. Этот пример отражает так называемый механизм раннего связывания.

4.2. ОБЪЕКТЫ В ТУРБО ПАСКАЛЕ

Описание объектов. Инкапсуляция. Для описания объектов в Турбо Паскале зарезервировано слово «object». Тип «Object» — это структура данных, которая содержит поля и методы. Описание объектного типа имеет следующий вид:

```
Type      <идентификатор типа объекта> = Object
    <поле>;
    ...
    <поле>;
    <метод>;
    ...
    <метод>;
End;
```

Поле содержит имя и тип данных. *Методы* — это процедуры или функции, объявленные внутри декларации объектного типа, в том числе и особые процедуры, создающие и уничтожающие объекты (конструкторы и деструкторы). Объявление метода внутри описания объектного типа состоит только из заголовка (как в разделе Interface в модуле).

Пример 4.1. Опишем объект «Обыкновенная дробь» с методами «НОД числителя и знаменателя», «Сокращение» и «Натуральная степень»:

```
Type Natur = 1..32767;
    Frac = Record P : Integer; Q : Natur End;
    Drob = Object A : Frac;
           Procedure NOD(Var C : Natur);
           Procedure Sokr;
           Procedure Stepen(N : Natur; Var C : Frac);
    End;
```

Описание объектного типа собственно и выражает такое свойство, как инкапсуляция.

Проиллюстрируем далее работу с описанным объектом, реализацией его методов и обращение к указанным методам (при этом понадобятся некоторые вспомогательные методы):

```
Type Natur = 1..MaxInt;
     Frac = Record P : Integer; Q : Natur End;
{----Описание объектного типа----}
     Drob = Object
            A : Frac;
            Procedure Vvod; {Ввод дроби}
            Procedure NOD(Var C : Natur); {НОД}
            Procedure Sokr;
            Procedure Stepen(N : Natur; Var C : Frac);
               Procedure Print; {Вывод дроби}
            End;
{----Описания методов объекта----}
Procedure Drob.NOD;
Var M, N : Natur;
```

```
Begin M := Abs(A.P); N := A.Q;
  While M <> N Do
  If M > N
  Then If M mod N <> 0 Then M := M mod N Else M := N
  Else If N mod M <> 0 Then N := N mod M Else N := M;
  C := M
End;
Procedure Drob.Sokr;
Var N : Natur;
Begin
  If A.P <> 0
  Then Begin
       Drob.NOD(N);
       A.P := A.P div N; A.Q := A.Q div N
  End
  Else A.Q := 1
End;
Procedure Drob.Stepen;
Var I : Natur;
Begin
  C.P := 1; C.Q := 1;
  For I := 1 To N Do
       Begin C.P := C.P * A.P; C.Q := C.Q * A.Q
       End;
End;
Procedure Drob.Vvod;
Begin
  Write('Введите числитель дроби: '); ReadLn(A.P);
  Write('Введите знаменатель дроби: '); ReadLn(A.Q);
End;
Procedure Drob.Print;
Begin WriteLn(A.P, '/', A.Q) End;
{----Основная программа----}
Var Z : Drob; F : Frac;
Begin
  Z.Vvod; {Ввод дроби}
  Z.Print; {Печать введенной дроби}
  Z.Sokr; {Сокращение введенной дроби}
  Z.Print; {Печать дроби после сокращения}
  Z.Stepen(4, F); {Возведение введенной дроби в
4-ю степень}
  WriteLn(F.P,'/', F.Q)
End.
```

Прокомментируем отдельные моменты в рассмотренном примере.

Во-первых, реализация методов осуществляется в разделе описаний после объявления объекта, причем при реализации метода достаточно указать его заголовок без списка параметров, но с указанием объектного типа, к которому он относится. Еще раз отметим, что все это напоминает создание модуля, когда ресурсы, доступные при его подключении, прежде всего объявляются в разделе Interface, а затем реализуются в разделе Implementation. В действительности объекты и их методы реализуют чаще всего именно в виде модулей.

Во-вторых, все действия над объектом выполняются только с помощью его методов.

В-третьих, для работы с отдельным экземпляром объектного типа в разделе описания переменных должна быть объявлена переменная (или переменные) соответствующего типа. Из примера видно, что объявление статических объектов не отличается от объявления других переменных, а их использование в программе напоминает использование записей.

Наследование. Объектные типы можно выстроить в иерархию. Один объектный тип может наследовать компоненты другого объектного типа. Наследующий объект называется «потомком», а объект, которому наследуют, — «предком». Если «предок» сам является чьим-либо «наследником», то «потомок» наследует и эти поля и методы. Следует подчеркнуть, что наследование относится только к типам, но не экземплярам объекта.

Описание типа «потомка» имеет отличительную особенность:

```
<имя типа потомка> = Object(<имя типа предка>);
```

Далее запись описания — обычная.

Следует помнить, что поля наследуются без какого-либо исключения, поэтому, объявляя новые поля, необходимо следить за уникальностью их имен, иначе совпадение имени нового поля с именем наследуемого поля вызовет ошибку. На методы это правило не распространяется, но об этом далее.

Пример 4.2. Опишем объектный тип «Вычислитель» с методами «Сложение», «Вычитание», «Умножение», «Деление» (моделирование некоторого исполнителя) и производный от него тип «Продвинутый вычислитель» с новыми методами «Степень», «Корень *n*-й степени»:

```
Type BaseType = Double;
     Vichislitel = Object
     A, B, C : BaseType;
     Procedure Init; {Ввод или инициализация полей}
     Procedure Slozh;
```

```
      Procedure Vich;
      Procedure Umn;
      Procedure Delen
End;
      NovijVichislitel = Object(Vichislitel)
      N : Integer;
      Procedure Stepen;
      Procedure Koren
End;
```

Обобщая сказанное ранее, перечислим правила наследования:

- информационные поля и методы «родительского» типа наследуются всеми его типами-«потомками» независимо от числа промежуточных уровней иерархии;
- доступ к полям и методам «родительских» типов в рамках описания любых типов-«потомков» выполняется таким образом, как будто они описаны в самом типе-«потомке»;
- ни в одном из типов-«потомков» не могут использоваться идентификаторы полей, совпадающие с идентификаторами полей какого-либо из «родительских» типов. Это правило относится и к идентификаторам формальных параметров, указанных в заголовках методов;
- тип-«потомок» может доопределить произвольное число собственных методов и информационных полей;
- любое изменение текста в «родительском» методе автоматически оказывает влияние на все методы типов-«потомков», которые его вызывают;
- в противоположность информационным полям идентификаторы методов в типах-«потомках» могут совпадать с именами методов в «родительских» типах. При этом одноименный метод в типе-«потомке» подавляет одноименный ему «родительский», и в рамках типа-«потомка» при указании имени такого метода будет вызываться именно метод типа-«потомка», а не «родительский».

Вызов наследуемых методов осуществляется согласно следующим принципам:

- при вызове метода компилятор сначала ищет метод, имя которого определено внутри типа объекта;
- если в типе объекта не определен метод с указанием его в операторе вызова метода, то компилятор в поисках метода с таким именем поднимается выше к непосредственному «родительскому» типу;

- если наследуемый метод найден и его адрес указан, следует помнить, что вызываемый метод будет работать так, как он определен и компилирован для «родительского» типа, а не для типа-«потомка». Если этот наследуемый «родительский» тип вызывает еще и другие методы, то вызываться уже будут только «родительские» или расположенные выше методы, так как вызов методов из расположенных ниже по иерархии типов не допускается.

Полиморфизм. Как уже отмечалось, полиморфизм (многообразие) предполагает определение класса или нескольких классов методов для родственных объектных типов таким образом, чтобы каждому классу отводилась своя функциональная роль. Методы одного класса обычно наделяются общим именем.

Пример 4.3. Пусть имеется «родительский» объектный тип «Выпуклый четырехугольник» (поля типа — «координаты вершин», заданные в порядке их обхода) и производные от него типы: «Параллелограмм», «Ромб» и «Квадрат». Описать для указанных фигур методы «Вычисление углов» (в градусах), «Вычисление диагоналей», «Вычисление длин сторон», «Вычисление периметра», «Вычисление площади».

Программа описания следующая:

```
Type BaseType = Double;
     FourAngle =  Object
                  x1, y1, x2, y2, x3, y3, x4, y4,
                  A, B, C, D, D1, D2,
                  Alpha, Beta, Gamma, Delta,
                  P, S : BaseType;
                  Procedure Init;
                  Procedure Storony;
                  Procedure Diagonali;
                  Procedure Angles;
                  Procedure Perimetr;
                  Procedure Ploshad;
                  Procedure PrintElements;
            End;
     Parall =     Object(FourAngle)
                  Procedure Storony;
                  Procedure Perimetr;
                  Procedure Ploshad;
            End;
     Romb = Object(Parall)
                  Procedure Storony;
                  Procedure Perimetr;
            End;
```

```
    Kvadrat =     Object(Romb)
                  Procedure Angles;
                  Procedure Ploshad;
            End;
Procedure FourAngle.Init;
Begin
    Write('Введите координаты вершин заданного четырехугольника: ');
    ReadLn(x1, y1, x2, y2, x3, y3, x4, y4);
End;
Procedure FourAngle.Storony;
Begin       A := Sqrt(Sqr(x2 - x1) + Sqr(y2 - y1));
            B := Sqrt(Sqr(x3 - x2) + Sqr(y3 - y2));
            C := Sqrt(Sqr(x4 - x3) + Sqr(y4 - y3));
            D := Sqrt(Sqr(x4 - x1) + Sqr(y4 - y1));
End;
Procedure FourAngle.Diagonali;
Begin
     D1 := Sqrt(Sqr(x1 - x3) + Sqr(y1 - y3));
     D2 := Sqrt(Sqr(x2 - x4) + Sqr(y2 - y4));
End;
Procedure FourAngle.Angles;
    Function Ugol(Aa, Bb, Cc : BaseType): BaseType;
    Var VspCos, VspSin: BaseType;
    Begin
           VspCos := (Sqr(Aa) + Sqr(Bb) -
                 Sqr(Cc))/(2 * Aa * Bb);
           VspSin := Sqrt(1 - Sqr(VspCos));
           If Abs(VspCos) > 1e-7
           Then Ugol := (ArcTan(VspSin/VspCos) +
                 Pi * Ord(VspCos < 0))/Pi * 180
           Else Ugol := 90
    End;
Begin Alpha := Ugol(D, A, D2);
      Beta := Ugol(A, B, D1);
      Gamma := Ugol(B, C, D2);
      Delta := Ugol(C, D, D1);
End;
Procedure FourAngle.Perimetr;
Begin P := A + B + C + D End;
Procedure FourAngle.Ploshad;
Var Per1, Per2 : BaseType;
Begin Per1 := (A + D + D2)/2; Per2 := (B + C + D1)/2;
      S := Sqrt(Per1*(Per1 - A) * (Per1 - D) *
      (Per1 - D2)) + Sqrt(Per2 * (Per2 - B) *
      (Per2 - C) * (Per2 - D1))
End;
Procedure FourAngle.PrintElements;
```

```
Begin
    WriteLn('Стороны: ', A:10:6, B:10:6, C:10:6, D:10:6,
'Углы: ', Alpha:10:4, Beta:10:4, Gamma:10:4, Delta:10:4,
'Периметр: ', P:10:6, 'Площадь: ', S:10:6, 'Диагонали:
', D1:10:6, D2: 10: 6)
End;
Procedure Parall.Storony;
Begin       A := Sqrt(Sqr(x2 - x1) + Sqr(y2 - y1));
            B := Sqrt(Sqr(x3 - x2) + Sqr(y3 - y2));
            C := A; D:= B
End;
Procedure Parall.Perimetr;
Begin P := 2 * (A + B) End;
Procedure Parall.Ploshad;
Var Per : BaseType;
Begin
    Per := (A + D + D2)/2;
     S := 2 * Sqrt(Per * (Per - A) * (Per - D) * (Per - D2))
End;
Procedure Romb.Storony;
Begin
    A := Sqrt(Sqr(x2 - x1) + Sqr(y2 - y1));
    B := A; C := A; D := A
End;
Procedure Romb.Perimetr;
Begin P := 2 * A End;
Procedure Kvadrat.Angles;
Begin       Alpha := 90; Beta := 90; Gamma := 90;
            Delta := 90;
End;
Procedure Kvadrat.Ploshad;
Begin S := Sqr(A) End;
{Основная программа}
Var obj : Kvadrat;
Begin
    obj.Init;
    obj.Storony;
    obj.Diagonali;
    obj.Angles;
    obj.Perimetr;
    obj.Ploshad;
    obj.PrintElements
End.
```

Таким образом, вычисление соответствующего элемента в фигуре, если это действие по сравнению с другими является уникальным, производится обращением к своему методу.

УПРАЖНЕНИЯ

1. Описать следующие объекты с указанием их методов:

а) «Телефонный звонок» — номер телефона, дата разговора, продолжительность, код города и т.д.;

б) «Поездка» — номер поезда, пункт назначения, дни следования, время прибытия, время стоянки и т.д.;

в) «Кинотеатр» — название кинофильма, сеанс, стоимость билета, количество зрителей и т.д.

Реализовать объявленные в объектном типе методы.

2. Дополнить объект Drob из примера 4.1 методами «Положительная дробь» (функция логического типа), «Правильная дробь» (функция логического типа), «Конечная десятичная дробь» (функция логического типа), «Период дроби» (существует, если соответствующая десятичная дробь не является конечной, в общем случае результат — «длинное» число). Реализовать и протестировать эти методы.

3. Описать объекты-«потомки» для объектов, указанных в первом упражнении. Реализовать объявленные в объектном типе методы.

4. Реализовать методы объекта «Вычислитель» из примера 4.2.

5. Дополнить набор объектов из примера 4.3 объектом «Прямоугольник», а также другими выпуклыми четырехугольниками, не являющимися параллелограммами (трапеция и т.д.), и реализовать соответствующие им методы.

6. Построить, последовательно применяя методы «Отобразить», «Сдвинуть», «Изменить размеры», «Спрятать», иерархии следующих объектов:

а) координаты — точка — горизонтальная линия — горизонтально-вертикальное перекрестье;

б) координаты — точка — наклонная под углом 45° линия — наклонное перекрестье;

в) координаты — точка — окружность — дуга (процедура Arc);

г) координаты — точка — эллипс (процедура FillEllipse) — эллиптическая дуга (процедура Ellipse);

д) координаты — точка — окружность — сектор (процедура PieSlice);

е) координаты — точка — сектор (процедура PieSlice) — эллиптическая дуга (процедура Ellipse);

ж) координаты — точка — прямоугольник (процедура Rectangle) — трехмерная полоса (процедура Bar3D);

з) координаты — точка — заштрихованный эллипс (процедура FillEllipse) — заштрихованный сектор (процедура Sector);

и) координаты — точка — окружность — заштрихованный сектор (процедура Sector);

к) координаты — точка — окружность — эллиптическая дуга (процедура Ellipse).

4.3. ИНТЕГРИРОВАННАЯ СРЕДА ПРОГРАММИРОВАНИЯ DELPHI

История и назначение Delphi. Delphi — интегрированная среда программирования, разработанная фирмой Borland, является результатом развития языка Турбо Паскаль, который, в свою очередь, представляет собой результат развития языка Паскаль. Изначально Паскаль был полностью процедурным языком. Турбо Паскаль, начиная с версии 5.5, обладает средствами объектно-ориентированного программирования. Языком программирования в Delphi представляет собой объектно-ориентированный язык Object Pascal, в основу которого положены конструкции Турбо Паскаля версии 7.0. В России Borland Delphi появился в конце 1993 г. и сразу же завоевал широкую популярность.

Среда программирования Delphi позволяет сравнительно легко и быстро создавать приложения операционной системы Windows, поэтому она получила название RAD (Rapid Application Development — среда быстрой разработки приложений).

Процесс разработки в Delphi предельно упрощен. И в первую очередь это относится к созданию интерфейса, на которое обычно приходится порядка 80 % времени разработки программы. Программисту необходимо просто поместить необходимые компоненты в окне Windows (в Delphi оно называется *формой*) и настроить их свойства с помощью специального инструмента — Object Inspector. С помощью данного инструмента можно связать *события* этих компонентов (нажатие кнопки, выбор мышью элемента в списке и т.д.) с процедурой обработки, и простое приложение готово. Причем разработчик имеет в своем распоряжении необходимые средства отладки (вплоть до пошагового выполнения команд процессора), удобную контекстную справочную систему, средства коллективной работы над проектом и т.д.

Delphi позволяет создавать распределенные Internet- и Intranet-приложения, используя для доступа к данным Borland DataBase Engine.

Язык Delphi Pascal (ранее — Object Pascal), используемый в Delphi, постоянно расширяется и дополняется компанией Borland, в полной мере поддерживая все требования, предъявляемые к объектно-ориентированному языку программирования. Как в строго типизированном языке, классы поддерживают только простое наследование, но зато интерфейсы могут иметь сразу несколько «предков». К числу особенностей языка следует отнести

поддержку обработки исключительных ситуаций (exceptions), а также перегрузку методов и подпрограмм (overload). Имеются также открытые массивы, варианты и вариантные массивы, позволяющие размещать в памяти любые структуры данных.

В Delphi можно создавать свои собственные компоненты, импортировать их, а также разрабатывать шаблоны проектов и мастеров, создающих заготовки проектов. Delphi предоставляет разработчику интерфейс для связи приложений (или внешних программ) с интегрированной оболочкой Delphi (IDE).

Интерфейс. Вид среды программирования Delphi отличается от вида многих других сред, которые можно увидеть в Windows, так как она следует спецификации, называемой SDI (Single Document Interface — однодокументный интерфейс), и состоит из нескольких отдельно расположенных окон.

Вид среды программирования Delphi непосредственно после запуска программы показан на рис. 4.1. В верхней строке, как и в любом приложении Windows, находится меню. Слева расположены дерево компонентов формы и инспектор компонентов, отображающий их свойства и события, по центру — собственно форма, а позади формы — редактор кода. Ниже строки меню располагаются панель инструментов и палитра компонентов.

Более детально элементы интерфейса Delphi показаны на рис. 4.2, а назначение некоторых кнопок панели инструментов поясняет рис. 4.3.

Форма. Имеет все признаки главного окна традиционных приложений Windows: размерную рамку, значок, заголовок, кнопки [Свернуть], [Развернуть], [Закрыть] (рис. 4.4) и управляется мышью.

Окно проектировщика форм (Form Designer) представляет собой заготовку, т.е. макет одного из окон разрабатываемого приложения. Форма является основным интерфейсным элементом в Delphi. Это визуальный компонент, присущий любой создаваемой в этой среде программе и исполняющий роль контейнера, который содержит другие компоненты, определяющие функциональность приложения.

Проектировщик форм позволяет во время разработки приложения выполнять следующие действия:

- добавлять компоненты в форму;
- модифицировать форму и ее компоненты;
- связывать обработчики событий компонента с программой на Delphi Pascal, содержащейся в редакторе кода.

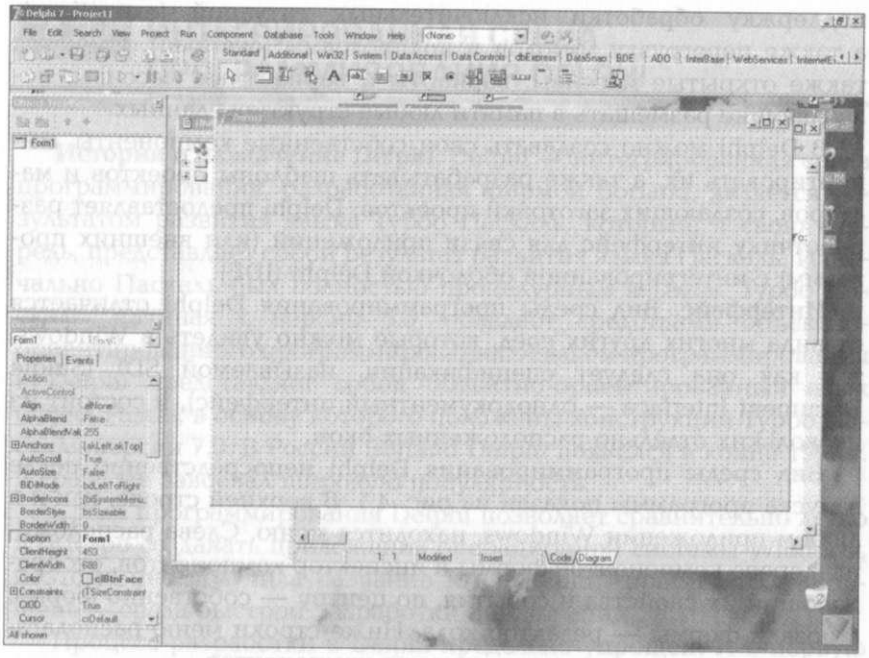


Рис. 4.1. Окно Delphi после запуска

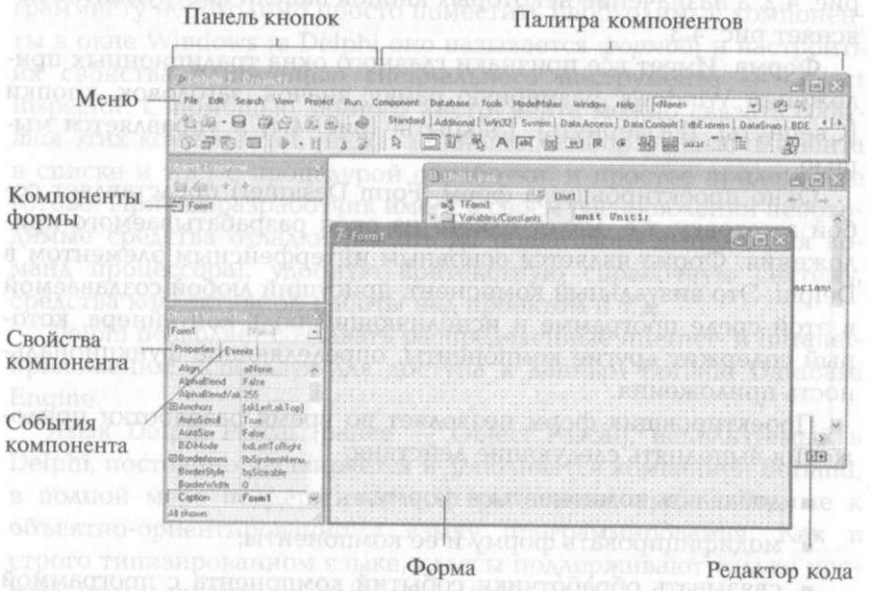


Рис. 4.2. Элементы интерфейса Delphi

Рис. 4.3. Назначение кнопок панели инструментов

В общих чертах процесс разработки программы на Delphi можно представить следующим образом: из области *Палитра компонентов* с помощью мыши надо выбрать компонент (кнопку, надпись, редактор текста и т.д.), поместить его в форму и задать значения свойств в области *Свойства*. Среда Delphi проанализирует

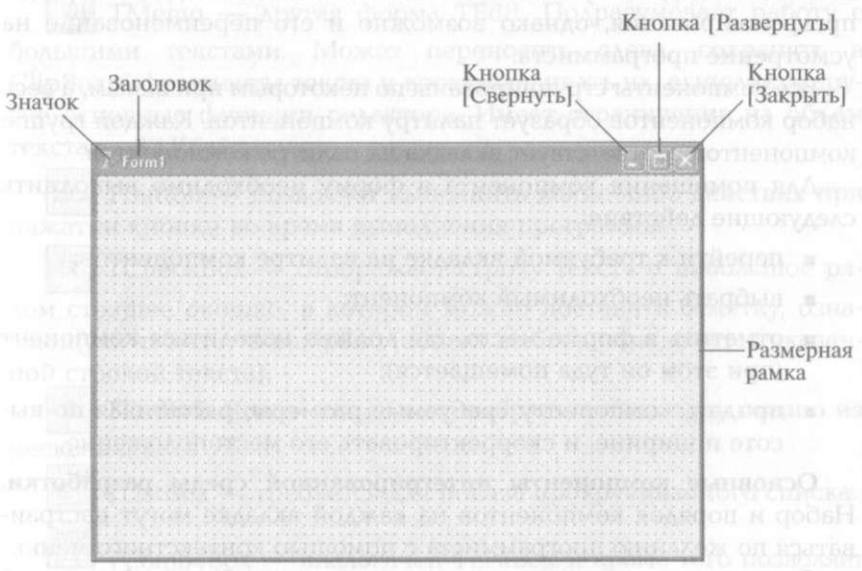


Рис. 4.4. Элементы формы

содержимое формы и создаст соответствующий программный код, а программисту останется только внести в него детали решения задачи — отклики на события.

4.4. КОМПОНЕНТЫ DELPHI. СВОЙСТВА КОМПОНЕНТОВ

Компонент Delphi — это функциональный элемент, обладающий набором свойств, определяющих его внешний вид и состояние, а также набором методов и событий, определяющих его поведение. Концепция использования компонентов при разработке программ напрямую связана с методологией объектно-ориентированного программирования. В данном случае с помощью компонентов происходит визуализация объектов, т.е. отображаются стандартные диалоговые окна, кнопки, списки и др. При этом каждый компонент предполагает собственный набор действий.

Все компоненты Delphi являются производными от класса TComponent, в котором инкапсулированы самые общие свойства и методы компонентов.

Идентификатор компонента строится по тем же правилам, что и идентификаторы других объектов Delphi, а также переменных, констант и т.д. По умолчанию имя компонента задается средой программирования, однако возможно и его переименование на усмотрение программиста.

Все компоненты сгруппированы по некоторым признакам, а весь набор компонентов образует палитру компонентов. Каждой группе компонентов соответствует вкладка на палитре компонентов.

Для помещения компонента в форму необходимо выполнить следующие действия:

- перейти к требуемой вкладке на палитре компонентов;
- выбрать необходимый компонент;
- отметить в форме место, где должен находиться компонент (при этом он туда помещается);
- придать компоненту требуемые размеры, растягивая по высоте и ширине, и скорректировать его местоположение.

Основные компоненты интегрированной среды разработки. Набор и порядок компонентов на каждой вкладке могут настраиваться по желанию программиста с помощью контекстного меню.

Опишем некоторые компоненты вкладки **Standard** типовой комплектации, показанной на рис. 4.5.

Рис. 4.5. Вкладка **Standard**

Frames — позволяет формировать на форме фреймы.

TMainMenu — позволяет поместить главное меню в разрабатываемую программу. Создание меню включает в себя три шага:

- помещение TMainMenu на форму;
- вызов Дизайнера меню через свойство Items в Инспекторе объектов;
- определение пунктов меню в Дизайнере меню.

TPopupMenu — позволяет создавать контекстные меню. У всех видимых объектов имеется свойство PopupMenu, где и указывается требуемое меню. Создается аналогично главному меню.

TLabel — служит для отображения текста на экране.

TEdit — стандартный управляющий элемент Windows для ввода. Может использоваться для отображения короткого фрагмента текста и позволяет пользователю вводить текст во время выполнения программы.

TMemo — другая форма TEdit. Подразумевает работу с большими текстами. Может переносить слова, сохранять в ClipBoard фрагменты текста и восстанавливать их, выполнять другие основные функции редактора. Имеет ограничения на объем текста — 32 Кбайт.

TButton — позволяет выполнить какие-либо действия при нажатии кнопки во время выполнения программы.

TCheckBox — отображает строку текста и небольшое рядом стоящее окошко, в котором можно поставить отметку, означающую, что что-то выбрано (объект выбора описывается указанной строкой текста).

TRadioButton — позволяет выбрать только одну опцию из нескольких.

TListBox — требуется для показа прокручиваемого списка.

TComboBox — аналогичен TListBox и кроме того позволяет водить информацию в поле ввода поверх TListBox. Имеется несколько типов TComboBox, но наиболее популярен спадающий вниз.

TScrollbar — полоса прокрутки, появляющаяся автоматически в объектах редактирования при необходимости прокрутки текста для просмотра.

Вкладка **Additional**, показанная на рис. 4.6, содержит дополнительные компоненты создания пользовательского интерфейса. Приведем описание некоторых из них.

TBitBtn — кнопка, аналогичная TButton, однако на ней можно разместить картинку (glyph). Имеет несколько предопределенных типов (bkClose, bkOK и др.), при выборе которых принимает соответствующий вид.

TTabSet — горизонтальные закладки, обычно используемые вместе с TNoteBook для создания многостраничных окон.

TNoteBook — используется совместно с TTabSet для создания многостраничного диалога, причем на каждой странице располагается свой набор объектов.

TOutline — используется для представления иерархических отношений связанных данных (например, дерева директорий).

Вкладка **System**, показанная на рис. 4.7, представляет собой набор компонентов для доступа к некоторым системным сервисам типа таймера, DDE, OLE и т. п.

Свойства компонентов и управление через свойства. Как уже отмечалось, свойства компонента определяют его вид и значения переменных его состояния. Очевидно, что перечень свойств многих компонентов, с одной стороны, повторяется, а с другой стороны, каждый компонент обладает собственным уникальным набором свойств.

Опишем свойства некоторых стандартных компонентов.

Компонент **Форма** (объект типа TForm) является основой программы. Свойства формы определяют вид окна программы (табл. 4.1).

Компонент **Label** предназначен для вывода текста на поверхность формы (табл. 4.2).

Рис. 4.6. Вкладка **Additional**

Рис. 4.7. Вкладка **System**

Таблица 4.1. Свойства формы

Свойство	Описание
Name	Имя формы. Используется в программе для управления формой и для доступа к ее компонентам
Caption	Текст заголовка
Top	Расстояние от верхней границы формы до верхней границы экрана
Left	Расстояние от левой границы формы до левой границы экрана
Width	Ширина формы
Height	Высота формы
ClientWidth, ClientHeight	Соответственно ширина и высота рабочей (клиентской) области формы, т. е. без учета ширины левой и правой границ
BorderStyle	Вид границы, которая может быть обычной (bsSizeable), тонкой (bsSingle) или отсутствовать (bsNone). Если у окна обычная граница, то во время работы программы пользователь может с помощью мыши изменить его размер. Изменить размер окна с тонкой границей нельзя. Если граница отсутствует, то на экран будет выведено окно без заголовка, положение и размер которого во время работы программы изменить нельзя
BorderIcons	Кнопки управления окном. Значение свойства определяет, какие кнопки управления окном будут доступны пользователю вовремя работы программы. Значение свойства задается посредством присвоения значений уточняющим свойствам: biSystemMenu, biMinimize, biMaximize и biHelp. Свойство biSystemMenu определяет доступность кнопки [Свернуть] и кнопки системного меню, biMinimize — доступность кнопки [Свернуть], biMaximize — доступность кнопки [Развернуть], biHelp — доступность кнопки вывода справочной информации
Icon	Значок в заголовке диалогового окна, обозначающий кнопку вывода системного меню
Color	Цвет фона, который можно задать, указав его название или привязку к текущей цветовой схеме операционной системы. В последнем случае цвет определяется текущей цветовой схемой, выбранным компонентом привязки и изменяется при изменении цветовой схемы операционной системы
Font	Шрифт, используемый по умолчанию компонентами, находящимися на поверхности формы
Canvas	Поверхность, на которую можно вывести графику

Таблица 4.2. Свойства текста

Свойство	Описание
Name	Имя компонента. Используется в программе для доступа к компоненту и его свойствам
Caption	Отображаемый текст
Left	Расстояние от левой границы поля вывода до левой границы формы
Top	Расстояние от верхней границы поля вывода до верхней границы формы
Height	Высота поля вывода
Width	Ширина поля вывода
AutoSize	Признак того, что размер поля определяется его содержимым
Wordwrap	Признак того, что слова, которые не помещаются в текущей строке, автоматически переносятся на следующую строку (значение свойства AutoSize должно быть False)
Alignment	Задает способ выравнивания текста внутри поля: по левому краю (taLeftJustify), по центру (taCenter) или по правому краю (taRightJustify)
Font	Шрифт, используемый для отображения текста. Уточняющие свойства определяют способ начертания символов (Font. Name), их размер (Font. Size) и цвет (Font. Color)
ParentFont	Признак наследования компонентом характеристик шрифта формы, на которой он находится. Если значение свойства равно True, то текст выводится шрифтом, установленным для формы
Color	Цвет фона области вывода текста
Transparent	Управляет отображением фона области вывода текста. Значение True делает область вывода текста прозрачной (область вывода не закрашивается цветом, заданным свойством Color)
Visible	Позволяет скрыть текст (False) или сделать его видимым (True)

Компонент Edit представляет собой поле ввода-редактирования строки символов (табл. 4.3).

Таблица 4.3. Свойства поля ввода-редактирования	
Свойство	Описание
Name	Имя компонента. Используется в программе для доступа к компоненту и его свойствам
Text	Текст, находящийся в поле ввода и редактирования
Left	Расстояние от левой границы компонента до левой границы формы
Top	Расстояние от верхней границы компонента до верхней границы формы
Height	Высота поля
Width	Ширина поля
Font	Шрифт, используемый для отображения вводимого текста
ParentFont	Признак наследования компонентом характеристик шрифта формы, на которой он находится. Если значение свойства равно True, то при изменении свойства Font формы автоматически изменяется значение свойства Font компонента
Enabled	Используется для ограничения возможности изменить текст в поле редактирования. Если значение свойства равно False, то текст в поле редактирования изменить нельзя
Visible	Позволяет скрыть компонент (False) или сделать его видимым (True)

Компонент **Button** представляет собой командную кнопку (табл. 4.4).

Компонент **Memo** представляет собой элемент редактирования текста, который может состоять из нескольких строк (табл. 4.5).

Таблица 4.4. Свойства командной кнопки	
Свойство	Описание
Name	Имя компонента. Используется в программе для доступа к компоненту и его свойствам
Caption	Текст на кнопке
Left	Расстояние от левой границы кнопки до левой границы формы
Top	Расстояние от верхней границы кнопки до верхней границы формы

Окончание табл. 4.4

Height	Высота кнопки
Width	Ширина кнопки
Enabled	Признак доступности кнопки. Если значение свойства равно True, то кнопка доступна. Если значение свойства равно False, то кнопка не доступна, например в результате щелчка мышью на кнопке событие Click не возникает
Visible	Позволяет скрыть кнопку (False) или сделать ее видимой (True)
Hint	Всплывающая подсказка-текст, который появляется рядом с указателем мыши при позиционировании указателя на командной кнопке (чтобы текст появился, значение свойства ShowHint должно быть True)
ShowHint	Разрешает (True) или запрещает (False) отображение подсказки при позиционировании указателя на кнопке

Таблица 4.5. Свойства Memo

Свойство	Описание
Name	Имя компонента. Используется для доступа к свойствам компонента
Text	Текст, находящийся в поле Memo. Рассматривается как единое целое
Lines	Массив строк, соответствующий содержимому поля. Доступ к строке осуществляется по номеру. Строки нумеруются с нуля
Lines.Count	Количество строк текста в поле Memo
Left	Расстояние от левой границы поля до левой границы формы
Top	Расстояние от верхней границы поля до верхней границы формы
Height	Высота поля
Width	Ширина поля
Font	Шрифт, используемый для отображения вводимого текста
ParentFont	Признак наследования свойств шрифта «родительской» формы

Компонент **RadioButton** представляет собой зависимую кнопку (табл. 4.6), состояние которой определяется состоянием других кнопок группы. Если в диалоговом окне надо организовать несколько групп переключателей, то каждую группу следует представить компонентом RadioGroup.

Компонент **CheckBox** представляет собой независимую кнопку — переключатель (табл. 4.7).

Компонент **ListBox** представляет собой список, в котором можно выбрать необходимый элемент (табл. 4.8).

Компонент **ComboBox** дает возможность ввести данные в поле редактирования посредством набора на клавиатуре или выбора из списка (табл. 4.9).

Компонент **BitBtn** так же, как и компонент Button является командной кнопкой. Однако он более универсален и главное может содержать картинку (табл. 4.10).

Компонент **ScrollBar** — полоса прокрутки, предназначенная для прокрутки содержимого окна и выбора значений из определенного интервала (табл. 4.11).

Таблица 4.6. Свойства радиокнопки

Свойство	Описание
Name	Имя компонента. Используется для доступа к свойствам компонента
Caption	Текст, который находится справа от кнопки
Checked	Состояние, вид кнопки: если кнопка выбрана, то Checked = True; если кнопка не выбрана, то Checked = False
Left	Расстояние от левой границы флажка до левой границы формы
Top	Расстояние от верхней границы флажка до верхней границы формы
Height	Высота поля вывода поясняющего текста
Width	Ширина поля вывода поясняющего текста
Font	Шрифт, используемый для отображения поясняющего текста
ParentFont	Признак наследования характеристик шрифта «родительской» формы

Таблица 4.7. Свойства переключателя

Свойство	Описание
Name	Имя компонента. Используется для доступа к свойствам компонента
Caption	Текст, который находится справа от флажка
Checked	Состояние, вид флажка: если флажок установлен, то Checked = True; если флажок сброшен, то Checked = = False
State	Состояние флажка. В отличие от свойства Checked, позволяет различать установленное, сброшенное и промежуточное состояния. Состояние флажка определяет одна из констант: cbChecked (установлен); cbGrayed (серый, неопределенное состояние); cbUnChecked (сброшен)
AllowGrayed	Определяет, может ли флажок быть в промежуточном состоянии: если AllowGrayed = False, то флажок может быть только установленным или сброшенным; если AllowGrayed = True, то допустимо промежуточное состояние
Left	Расстояние от левой границы флажка до левой границы формы
Top	Расстояние от верхней границы флажка до верхней границы формы
Height	Высота поля вывода поясняющего текста
Width	Ширина поля вывода поясняющего текста
Font	Шрифт, используемый для отображения поясняющего текста
ParentFont	Признак наследования характеристик шрифта «родительской» формы

Таблица 4.8. Свойства списка

Свойство	Описание
Name	Имя компонента. В программе используется для доступа к компоненту и его свойствам
Items	Элементы списка — массив строк
Count	Количество элементов списка

Окончание табл. 4.8

Свойство	Описание
Sorted	Признак необходимости автоматической сортировки (True) списка после добавления очередного элемента
ItemIndex	Номер выбранного элемента (элементы списка нумеруются с нуля). Если в списке ни один из элементов не выбран, то значение свойства равно –1
Left	Расстояние от левой границы списка до левой границы формы
Top	Расстояние от верхней границы списка до верхней границы формы
Height	Высота поля списка
Width	Ширина поля списка
Font	Шрифт, используемый для отображения элементов списка
ParentFont	Признак наследования свойств шрифта «родительской» формы

Таблица 4.9. Свойство ComboBox

Свойство	Описание
Name	Имя компонента. Используется для доступа к свойствам компонента
Text	Текст, находящийся в поле ввода-редактирования
Items	Элементы списка — массив строк
Count	Количество элементов списка
ItemIndex	Номер элемента, выбранного в списке. Если ни один из элементов списка не был выбран, то значение свойства равно –1
Sorted	Признак необходимости автоматической сортировки (True) списка после добавления очередного элемента
DropDownCount	Количество отображаемых элементов в раскрытом списке. Если количество элементов списка больше, чем DropDownCount, то появляется вертикальная полоса прокрутки

Окончание табл. 4.9

Свойство	Описание
Left	Расстояние от левой границы компонента до левой границы формы
Top	Расстояние от верхней границы компонента до верхней границы формы
Height	Высота компонента (поля ввода-редактирования)
Width	Ширина компонента
Font	Шрифт, используемый для отображения элементов списка
ParentFont	Признак наследования свойств шрифта «родительской» формы

Таблица 4.10. Свойства BitBtn

Свойство	Описание
Name	Имя компонента. Используется в программе для доступа к компоненту и его свойствам
Caption	Текст, отображаемый на кнопке
Font	Шрифт, который используется для отображения текста
Left	Расстояние от левой границы формы до левой границы компонента
Top	Расстояние от верхней границы формы до верхней границы компонента
Width	Ширина поля компонента
Height	Высота поля компонента
Enabled	Признак доступности кнопки: кнопка доступна, если значение свойства равно True, и недоступна, если оно равно False
Visible	Признак видимости кнопки на поверхности формы: если значение свойства равно True — кнопка отображается, в противном случае она невидима
Glyph	Картинка, отображаемая на кнопке (файл изображения)
NumGlyphs	Количество картинок в файле изображения, указанного в свойстве Glyph

Окончание табл. 4.10

Свойство	Описание
Layout	Определяет взаимоположение картинки и текста на кнопке. Может принимать следующие значения: blGlyphLeft — картинка располагается слева от надписи, blGlyphRight — справа, blGlyphTop — сверху, blGlyphBottom — снизу
Spacing	Расстояние от картинки до надписи. Расстояние задается в пикселах

Таблица 4.11. Свойства полосы прокрутки

Свойство	Описание
Name	Имя компонента. Используется в программе для доступа к компоненту и его свойствам
Enabled	Признак доступности компонента. Прокрутка доступна, если значение свойства равно True, и недоступна, если оно равно False
Kind	Определяет положение прокрутки: горизонтальное или вертикальное
Max	Задает максимальное значение положения ползунка
Min	Задает минимальное значение положения ползунка
Visible	Признак видимости компонента на поверхности формы. Прокрутка отображается, если значение свойства равно True. В противном случае она невидима
LargeChange	Указывает, на какое количество точек перемещается ползунок при щелчке мышью по полосе либо при нажатии клавиши [PageUp] и [PageDown]
Left	Расстояние от левой границы формы до левой границы компонента
Top	Расстояние от верхней границы формы до верхней границы компонента
Width	Ширина поля компонента
Height	Высота поля компонента

Свойства компонентов можно изменять на вкладке **Properties** в Object Inspector. Изменение свойств компонента Form представлено на рис. 4.8.

Демонстрация настройки свойств компонентов будет приведена далее в интегрированном примере.

События компонентов и определение процедур на основе событий. Как уже говорилось, события определяют поведение компонента во время работы программы. Предполагается, что реакцией на действия пользователя будет исполнение того или иного программного кода. Следовательно, при составлении сценария программы необходимо учесть все предполагаемые действия пользователя и предусмотреть для каждого случая адекватную реакцию.

События компонента можно программировать на вкладке **Events** в Object Inspector. Изменение реакции на события для компонента Form представлено на рис. 4.9.

Двойной щелчок левой клавишей мыши инициирует редактирование кода для соответствующего события. При этом шаблон метода создается автоматически.

Кроме собственно выполнения некоторого действия при обработке события могут быть получены какие-то результаты, которые могут передаваться для дальнейшего использования любому объекту (компоненту) программы.

Программа на Delphi. Любая программа в среде Delphi состоит из файла проекта (файл с расширением.dpr) и одного или нескольких модулей. Каждый из таких файлов составляет программную единицу Delphi.

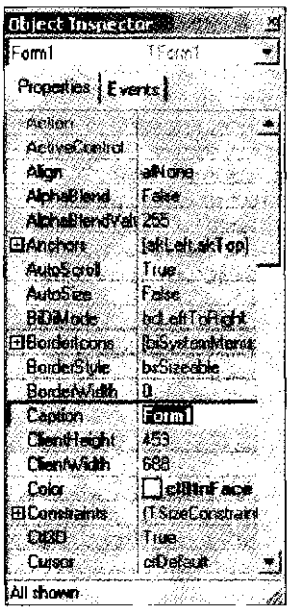


Рис. 4.8. Окно инспектора объектов

Файл проекта для каждой программы имеет примерно одинаковый вид. Например:

```
Program Project1;
Uses Forms, Unit1 In 'Unit1.pas'
{Form1};
    {$R *.res}
Begin
    Application.Initialize;
    Application.
CreateForm(TForm1, Form1);
    Application.Run;
End.
```

В объекте Application собраны данные и подпрограммы, необходимые для нор-

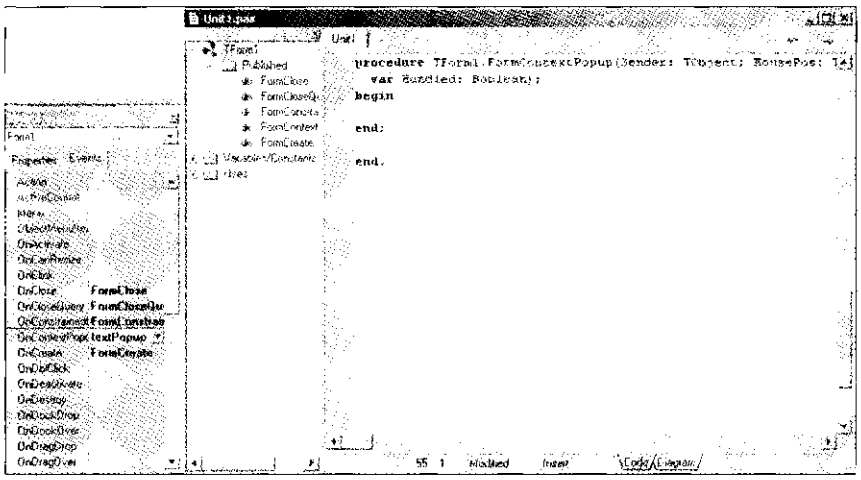


Рис. 4.9. Окно программного модуля для обработки события

мального функционирования Windows-программы в целом. Delphi автоматически создает объект-программу Application для каждого нового проекта.

Содержание и функциональное назначение методов, вызываемых в программе, соответствует их названию. Так, метод Initialize инициализирует программу. Метод CreateForm объекта Application создает и показывает на экране окно главной формы, а метод Run реализует бесконечный цикл получения и обработки поступающих от Windows сообщений о действиях пользователя. Когда пользователь щелкнет мышью на кнопке Close, Windows передаст программе специальное сообщение, которое заставит программу прекратить работу и освободить назначенные ей системные ресурсы. Файл проекта полностью формируется самой средой Delphi и в большинстве случаев не предназначен для редактирования (по крайней мере, в учебных задачах).

Несколько слов об отличии языка программирования Delphi от языка Turbo Pascal: к простым типам данных здесь добавляется тип дата-время (средства для работы с этим типом представлены в табл. П3.1 приложения), к сложным — процедурные типы, варианты и классы.

Особую роль в Delphi играют строки, и так как здесь для ввода-вывода (обмена данными между компонентами) чаще всего используется именно строковый тип данных, набор необходимых для работы соответствующих средств приведен в табл. П3.2 приложения.

4.5. СОБЫТИЙНО-УПРАВЛЯЕМОЕ ПРОГРАММИРОВАНИЕ

В Delphi программный код хранится в виде модулей. При этом структура модуля достаточно близка к аналогичной структуре в Турбо Паскале, хотя имеются и некоторые отличия:

```
Unit <имя>;
Interface
    <интерфейсная часть>
Implementation
    <исполняемая часть>
Initialization
    <инициализирующая часть>
Finalization
    <завершающая часть>
End.
```

Здесь **Unit** — зарезервированное слово (единица), начинающее заголовок модуля; <имя> — имя модуля (правильный идентификатор); **Interface** — зарезервированное слово (интерфейс), начинающее интерфейсную часть модуля; **Implementation** — зарезервированное слово (выполнение), начинающее исполняемую часть; **Initialization** — зарезервированное слово (инициализация), начинающее инициализирующую часть модуля; **Finalization** — зарезервированное слово (завершение), начинающее завершающую часть модуля; **End** — зарезервированное слово, являющееся признаком конца модуля.

Таким образом, модуль состоит из заголовка и четырех составных частей, любая из которых может быть пустой.

Назначение первых двух частей модуля такое же, как в Турбо Паскале, поэтому не будем останавливаться на них подробно. Инициализирующая и завершающая части модуля чаще всего отсутствуют вместе с начинающим их словами **Initialization** и **Finalization**. В инициализирующей части размещаются операторы, которые исполняются до передачи управления основной программе и обычно используются для подготовки ее работы. Например, в них могут инициализироваться переменные, открываться необходимые файлы и др. В завершающей части указываются операторы, выполняющиеся после завершения работы основной программы. Если несколько модулей содержат инициализирующие части, эти части выполняются последовательно друг за дру-

гом в порядке перечисления модулей в **Uses** главной программы. Если несколько модулей содержат завершающие части, эти части выполняются последовательно друг за другом в порядке, обратном перечислению модулей в **Uses** главной программы.

Рассмотрим использование некоторых стандартных компонентов.

Для примера разработаем программу, которая позволяет решать линейное уравнение вида $ax + b = 0$.

Спланируем реализуемый проект. Очевидно, что форма для решения задачи должна содержать заголовок с текстом решаемой задачи. Кроме того, должны иметься текстовые поля ввода для значений *a*, *b*. Результат также должен выводится в определенное поле. Для получения результата после ввода данных необходимо на форме разместить соответствующую кнопку. Для инициализации переменных и очистки формы от ранее полученных результатов требуется кнопка сброса.

Примечание. Будем полагать, что пользователи вводят корректные данные. Отметим также, что в отличие от Турбо Паскаля разделителем целой и дробной частей вещественного числа в русскоязычной версии Windows (если не выбрана другая настройка) является запятая.

Примерный вид окна программы решения указанной задачи показан на рис. 4.10.

Определим, какие компоненты необходимо помещать на форму. Для вывода сообщения «Программа решения линейного уравнения ax + b = 0», надписей «a», «b» и ответа используем компонент Label. Для ввода информации используем компонент Edit, и, наконец, для кнопок — компонент Button.

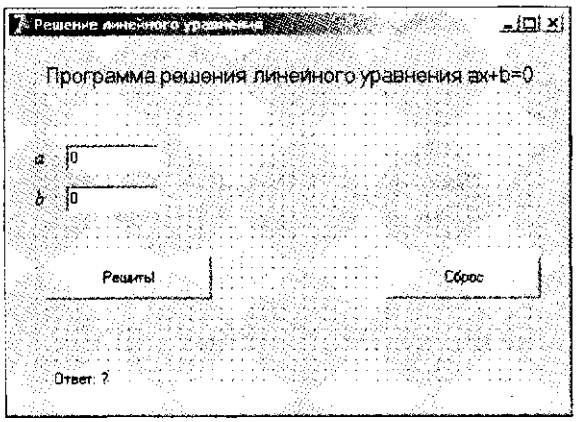


Рис. 4.10. Форма интерфейса приложения

Итак, поместим все необходимые компоненты на форму, отрегулируем их размеры, установим значения свойств по умолчанию и переименуем задаваемые по умолчанию идентификаторы компонентов для лучшей читаемости кода программы.

Идентификатор компонента для кнопки [Решить!] назовем Solution, для кнопки [Сброс] — Clear, поле вывода ответа — Answer. Остальные идентификаторы оставим неизменными.

Для всех компонентов установим необходимое значение свойства Caption для отображения запланированного текста.

Далее программируем реакцию на действия пользователя при работе программы. В нашем случае следует предусмотреть реакцию на нажатие кнопок [Решить!] и [Сброс].

Код для кнопки [Сброс]:

```
Procedure TForm1.ClearClick(Sender : TObject);
Begin
  Edit1.Text := '0';
  Edit2.Text := '0';
  Answer.Caption := 'Ответ:?';
End;
```

Код для кнопки [Решить!]:

```
Procedure TForm1.SolutionClick(Sender : TObject);
Var a, b : real; x : real; s : String;
Begin
  a := StrToFloat(Edit1.Text);
  b := StrToFloat(Edit2.Text);
  If a <> 0
  Then Begin
            x := -b/a;
            s := 'Единственное решение: ';
     End
  Else If a = 0
       Then If b = 0
            Then s := 'Бесконечное множество решений'
            Else s := 'Нет решений';
  If a<>0
  Then Answer.Caption := 'Ответ: ' + s +
FloatToStr(x)
  Else Answer.Caption := 'Ответ: ' + s;
End;
```

Обратим внимание на приведение типов в последнем коде. В поле редактирования данные заносятся в текстовом формате, и

для дальнейшей обработки требуется перевод их в числовой формат. Для вывода данных, наоборот, требуются только строковые величины.

Данная программа позволяет решать заданное линейное уравнение для любых вещественных данных. Для ее тестирования необходим набор, по крайней мере, из трех тестов, обеспечивающих проверку всех разветвлений.

Полный код модуля для реализуемого проекта имеет следующий вид:

```
Unit Unit1;
Interface
Uses
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
Type
  TForm1 = Class(TForm)
             Label1 : TLabel;
             Label2 : TLabel;
             Edit1 : TEdit;
             Label3 : TLabel;
             Edit2 : TEdit;
             Solution : TButton;
             Clear : TButton;
             Answer : TLabel;
  Procedure SolutionClick(Sender : TObject);
  Procedure ClearClick(Sender : TObject);
  Private
       {Private declarations}
  Public
       {Public declarations}
  End;
Var Form1 : TForm1;
Implementation
  {$R *.dfm}
Procedure TForm1.SolutionClick(Sender : TObject);
Var a, b : Real; x : Real; s : String;
Begin
  a := StrToFloat(Edit1.Text);
  b := StrToFloat(Edit2.Text);
  If a <> 0
  Then Begin
            x := -b/a;
```

```
            s := 'Единственное решение: ';
        End
    Else If a = 0
        Then If b = 0
            Then s := 'Бесконечное множество решений'
            Else s := 'нет решений';
    If a <> 0
    Then Answer.Caption := 'Ответ: ' + s +
    FloatToStr(x)
    Else Answer.Caption := 'Ответ: ' + s;
End;
Procedure TForm1.ClearClick(Sender: TObject);
Begin
        Edit1.Text := '0';
        Edit2.Text := '0';
        Answer.Caption := 'Ответ:?';
    End;
End.
```

Данная программа готова к работе. Компиляция осуществляется следующим образом:

- выполнить команду *Проект/Опции* (Project/Option), после чего откроется диалоговое окно *Опции проекта* (Project Options);
- выбрать вкладку *Каталог/Условия* (Directories/Conditionals);
- задать в разделе Каталоги в строке Результат (Output directory) путь до папки с файлами проекта, в которой будет записан результирующий exe-файл;
- выполнить команду *Проект/Компилировать* проект (Project/Build All).

Имя exe-файла совпадает с именем файла проекта. Этот файл можно будет запустить вне среды Delphi. Если exe-файл не требуется, то строку *Результат* (Output directory) следует очистить, в этом случае файл будет компилироваться в память.

В процессе отладки для запуска программы можно использовать кнопку ▶ или клавишу F9.

УПРАЖНЕНИЯ

1. На основе приложения для решения линейного уравнения разработать:

а) приложение для решения квадратного уравнения;

б) приложение для решения системы из двух линейных уравнений с двумя неизвестными.

2. Дополнить приложение для решения линейного уравнения и приложения из первого упражнения проверкой ввода данных.

3. Написать приложение для второго упражнения к подразд. 4.2 (для работы с обыкновенными дробями).

4.6. ТЕХНОЛОГИЯ СОЗДАНИЯ ПРИЛОЖЕНИЙ В DELPHI

Объектно-ориентированный подход, так же как любая парадигма программирования, подразумевает прохождение ряда этапов при разработке программного продукта. Ранее уже описывались этапы разработки программы при использовании структурного подхода к программированию, а также методика объектно-ориентированного проектирования. Безусловно, использование объектов определяет некоторую специфику в последовательности действий при создании приложения, хотя многое из того, что обсуждалось и использовалось ранее, остается в силе. Заметим, что написание учебного приложения — это моделирование действий по разработке «взрослых» программных продуктов, подразумевающее некоторое упрощение процесса с сохранением его основных черт. С этих позиций и будем рассматривать далее этапы разработки приложений в Delphi. Отметим также, что процесс создания программы чаще всего носит итерационный характер, т.е. сначала выполняются анализ, проектирование и реализация интерфейса или его части, а затем программа наращивается до получения окончательного варианта.

В разработанной в подразд. 4.5 программе большинство из этих этапов имеется. Остановимся на них подробнее.

Этапы разработки приложений. Упрощенное представление этапов разработки:

- постановка задачи, изучение предметной области, построение модели (математической, информационной), проецирование построенной модели на разрабатываемое ООП-приложение;
- проектирование ООП-приложения;
- разработка интерфейса пользователя;
- программирование приложения, тестирование и отладка программного кода;
- разработка документации.

Не будем подробно останавливаться на первом этапе, поскольку его содержание, детально описанное при рассмотрении структурного подхода к программированию, не имеет принципиальных отличий при ОПП.

Более детально обсудим последующие этапы разработки ООП-приложения в визуальной среде.

Проектирование ООП-приложения. Результатом первого этапа разработки является информационная модель исследуемого процесса, явления (в нашем случае решаемой содержательной задачи). По сути, эта модель уже включает в себя, как минимум, набор объектов и связей между ними (т.е. для исходной системы выполнена декомпозиция). На этапе проектирования ООП-приложения каждому объекту необходимо дать уникальное имя (задать идентификатор). Здесь же принимается решение, какие из объектов требуют визуализации, т.е. в терминах Delphi, какие компоненты будут отображены на форме.

Для каждого выделенного объекта необходимо указать перечень характеристик (свойств) и действий над ним. Действия над объектом отображаются посредством набора методов. Далее будет показано, что чаще всего в качестве такого рода действий в ООП-приложении, разрабатываемом для функционирования в операционной системе с графическим интерфейсом (например, Windows), выступают реакции на разного рода события, как внешние, так и внутренние.

Разработка интерфейса пользователя. При разработке интерфейса пользователя необходимо отталкиваться от функционального назначения приложения. В Delphi особенность разработки интерфейса заключается в том, что многие его элементы (форма, компоненты и т.д.) уже присутствуют в среде программирования. Таким образом, интерфейс создается на базе имеющихся «кирпичиков» и адаптируется под запросы потенциального конечного пользователя.

При разработке интерфейса пользователя необходимо соблюдать ряд правил.

1. При проектировании окон (форм) ввода данных:

- для команд всегда следует создавать клавишные эквиваленты (при этом необходимо там, где это уместно, сохранять привычные эквиваленты), не заставляя пользователя применять исключительно мышь;
- расположение элементов на главной форме должно быть согласовано с задачами пользователя;

- при вводе данных должна присутствовать заметная, но ненавязчивая связь с пользователем (например, синтаксический контроль и, если возможно, исправление ошибок и опечаток);
- для нескольких разных форм ввода не следует использовать существенно отличающиеся интерфейсы.

2. При создании меню:

- следовать стандартным соглашениям о расположении пунктов меню, принятым в Windows;
- группировать пункты меню в логическом порядке и по содержанию;
- для группировки пунктов в раскрывающихся меню использовать разделительные линии;
- избегать избыточных меню;
- использовать клавиатурные эквиваленты команд и «горячие» клавиши;
- помещать на панель инструментов часто используемые команды меню.

3. При работе приложения в процессе ожидания следует информировать пользователя о ходе работы (например, с помощью индикатора состояния выполнения задания).

Естественными являются общие дизайнерские требования: сочетание цвета фона приложения и цвета выводимого на него текста, количество используемых цветов и др.

Программирование, тестирование, отладка. Как уже отмечалось, на этапе проектирования приложения выделяется функциональное назначение всех его элементов и возможные реакции на внешние и внутренние события.

Часть программного кода, которая касается интерфейса приложения, генерируется автоматически при помещении и настройке свойств соответствующих элементов. Часть кода, касающуюся обработки событий, создает программист. Кроме того, программный код может содержать вспомогательные подпрограммы, необходимые для функционирования отдельных частей или приложения в целом. Размещение кода таких подпрограмм будет рассмотрено далее.

Очевидно, что каждый из разработанных методов и вспомогательных подпрограмм, а затем и комплекс в целом подвергаются процедуре тестирования. Приемы тестирования мало отличаются от приемов, применяемых для тестирования кода структурных программ. Для каждого метода или подпрограммы составляется столь-

ко тестов, сколько возможно вариантов прохождения данного метода, проверяются граничные и недопустимые состояния. При этом требования к тестовым наборам сохраняются. В коде программы необходимо предусмотреть верификацию данных при вводе.

Далее тестируется взаимодействие разных элементов программы и приложения в целом. Отметим, что тестовые наборы разрабатываются до начала процедуры тестирования. В общем случае они являются частью документации приложения. В настоящее время при разработке программного обеспечения используют системы автоматической генерации тестовых наборов, которые создаются на основе модели решаемой задачи. В учебном программировании тестовые наборы создаются «вручную».

Разработка документации. Любой программный продукт сопровождается документацией. При разработке ООП-приложения состав и назначение документации должен отвечать тем же требованиям, что и при разработке программного обеспечения для других парадигм программирования. Минимальный пакет документации составляют руководство программиста и руководство пользователя. Состав и структура документации в общем случае стандартизированы.

Важно отметить наличие самодокументирования программного кода, т. е. комментариев к его блокам в тексте программы.

При разработке учебных программ рекомендуется следующий состав отчетной документации:

- описание постановки задачи;
- описание модели;
- описание проекта (количество элементов на форме, их функции, взаимодействие между собой, взаимодействие с другими приложениями) и инициализация свойств объектов;
- тестовые наборы для каждого метода, вспомогательных подпрограмм, интерфейсной части и приложения в целом;
- программный код с комментариями;
- результаты работы программы (при выполнении расчетов или каких-либо других действий, предполагающих эти результаты).

4.7. ПРИМЕРЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ DELPHI

Рассмотрим примеры прохождения этапов разработки программного продукта.

Пример 4.4. Требуется запрограммировать калькулятор для выполнения действий с числами в системе счисления с основанием p ($2 \le p \le 9$).

Спланируем реализуемый проект и примерный интерфейс приложения. Форма будет содержать кнопки для набора цифр. Причем, если какие-либо цифры не входят в систему счисления с текущим основанием, они скрываются. Выбор основания системы счисления будем осуществлять из раскрывающегося списка, который имеется на форме. Результаты (как промежуточные, так и итоговые) выводятся в определенное поле. Еще четыре кнопки обозначим знаками четырех арифметических операций, также необходимы кнопка [=] и кнопка сброса [C].

Опишем предполагаемое функционирование приложения:

- при нажатии цифровой кнопки очередная цифра отображается в записи текущего числа в поле результата;
- при нажатии кнопки [C] в поле результата отображается нуль;
- при нажатии кнопок со знаками операций текущее значение используется в качестве второго операнда (предполагается, что первый уже имеется в памяти или, по крайней мере, равен нулю) в той операции, которая была выбрана ранее, результат и текущая выбранная операция запоминаются, в поле результата индицируется нуль (поле готово к вводу следующего числа);
- при нажатии кнопки [=] вычисляется окончательный результат в соответствии с последней выбранной операцией;
- при выборе основания системы счисления значение в поле результата переводится в эту систему счисления.

Вид окна программы решения указанной задачи показан на рис. 4.11.

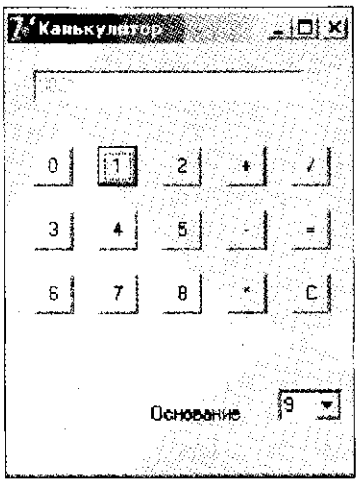


Рис. 4.11. Интерфейс приложения «Калькулятор»

Приведем некоторые листинги (другие предлагается разработать самостоятельно по аналогии).

Код процедуры для события «Выбор основания системы счисления» (с использованием компонента ComboBox) следующий:

```
Procedure TCalculator.OsnovanieSelect(Sender:
TObject);
Var a, p : Integer; s : String;
Begin
p := pOld;
  _PTo10(Rezalt.Text, p, a);
  pOld := StrToInt(Osnovanie.Text);
  _10toP(a, pOld, s);
  Rezalt.Text := s;
  Case StrToInt(Osnovanie.Text) Of
   2: Begin
      _2.Visible := False;
      _3.Visible := False;
      _4.Visible := False;
      _5.Visible := False;
      _6.Visible := False;
      _7.Visible := False;
      _8.Visible := False;
      End;
   3: Begin
      _2.Visible := True;
      _3.Visible := False;
      _4.Visible := False;
      _5.Visible := False;
      _6.Visible := False;
      _7.Visible := False;
      _8.Visible := False;
      End;
   4: Begin
      _2.Visible := True;
      _3.Visible := True;
      _4.Visible := False;
      _5.Visible := False;
      _6.Visible := False;
      _7.Visible := False;
      _8.Visible := False;
      End;
```

```
5: Begin
    _2.Visible := True;
    _3.Visible := True;
    _4.Visible := True;
    _5.Visible := False;
    _6.Visible := False;
    _7.Visible := False;
    _8.Visible := False;
   End;
6: Begin
    _2.Visible := True;
    _3.Visible := True;
    _4.Visible := True;
    _5.Visible := True;
    _6.Visible := False;
    _7.Visible := False;
    _8.Visible := False;
   End;
7: Begin
    _2.Visible := True;
    _3.Visible := True;
    _4.Visible := True;
    _5.Visible := True;
    _6.Visible := True;
    _7.Visible := False;
    _8.Visible := False;
   End;
8: Begin
    _2.Visible := True;
    _3.Visible := True;
    _4.Visible := True;
    _5.Visible := True;
    _6.Visible := True;
    _7.Visible := True;
    _8.Visible := False;
   End;
9: Begin
    _2.Visible := True;
    _3.Visible := True;
    _4.Visible := True;
    _5.Visible := True;
    _6.Visible := True;
    _7.Visible := True;
```

```
      _8.Visible := True;
      End;
  End;
End;
```

Щелчок мышью по кнопке с обозначением цифры на примере кнопки [0]:

```
Procedure TCalculator._0Click(Sender : TObject);
Begin
  If Rezalt.Text = '0' Then Rezalt.Text :=
_0.Caption
  Else Rezalt.Text := Rezalt.Text + _0.Caption
End;
```

Щелчок мышью по кнопке со знаком операции на примере операции [+]:

```
Procedure TCalculator.PlusClick(Sender : TObject);
Begin
  Operation;
  Znak := '+';
End;
```

Щелчок мышью по кнопке [=]:

```
Procedure TCalculator.Button1Click(Sender :
TObject);
Var s : String;
Begin
  Operation;
  Znak := ' ';
  _10ToP(a, POld, s);
  Rezalt.Text := s;
  a := 0;
End;
```

Приведем также реализацию вспомогательных процедур, которые непосредственно выполняют переводы из десятичной системы счисления в систему с основанием *p*, и наоборот, а также собственно выполнение арифметических операций:

```
Procedure _10ToP(a, p : Integer; Var s : String);
Begin
  If a = 0 Then s :='0' Else s :='';
  While a <> 0 Do
  Begin
```

```
        s := IntToStr(a mod p) + s;
        a := a div p
    End;
End;
Procedure _PTo10(s : String; p : Integer; Var a :
Integer);
Var i : Integer;
Begin
    a := 0;
    For  i := 1 To length(s) Do
        a := a * p + StrToInt(s[i])
End;
Procedure Operation;
Var r : Integer;
Begin
    _pTo10(Calculator.Rezalt.Text, POld, r);
    If a = 0 Then a := r
    Else Case Znak Of
        '+': a := a + r;
        '-': a := a - r;
        '*': a := a * r;
        '/': a := a div r
        End;
    Calculator.Rezalt.Text := '0';
End;
```

Указанные процедуры не входят в состав основных и присутствуют лишь в разделе реализации модуля проекта (Implementation).

Наконец, при запуске приложения инициализируются переменные (глобальные и некоторые поля компонентов):

```
Procedure TCalculator.FormCreate(Sender :
TObject);
Begin
    _2.Visible := False;
    _3.Visible := False;
    _4.Visible := False;
    _5.Visible := False;
    _6.Visible := False;
    _7.Visible := False;
    _8.Visible := False;
    pOld := 2; a := 0;
End;
```

Пример 4.5. Требуется реализовать приложение, моделирующее следующую игру для младших школьников. На поле есть две коробки: для черных и для белых шариков, а также некоторое количество черных и белых шариков (от 2 до 10). Разложить эти шарики по коробкам и указать, каких больше — черных или белых.

При реализации требуемого приложения будем использовать специальный прием связывания программ с данными в Windows, который называется Drag&Drop (перетащи и отпусти). Использование соответствующих событий и свойств данного механизма применительно к объектам Delphi опишем по ходу реализации.

Спланируем проект и примерный интерфейс приложения. Форма будет содержать две коробки (компоненты TLabel) и десять шариков (компоненты TShape). При этом при каждом запуске программы число шариков и их цвет будут задаваться случайным образом.

Опишем предполагаемое функционирование приложения:

- при запуске инициализируются необходимые свойства компонентов и задаются число и цвет шариков, а лишние шарики при этом скрываются;
- при перетаскивании шарика в «свою» коробку и отпускании клавиши мыши шарик исчезает и счетчик числа шариков в коробке увеличивается на единицу;
- при перетаскивании шарика в «чужую» коробку его размещение в ней запрещается и шарик остается на месте.

Примерный вид формы для создаваемого приложения показан на рис. 4.12.

Возможный результат работы этой программы показан на рис. 4.13.

Опишем логику работы создаваемого приложения на основе программного кода.

При запуске приложения необходимо инициализировать все свойства и разместить случайное число шариков на форме.

Программа данной процедуры следующая:

```
Procedure TForm1.FormCreate(Sender : TObject);
Var i : Integer;
Begin
  b := 0; w := 0;
  Black.Height := 80; Black.Width := 80;
  Black.Hint := 'b';
  White.Height := 80; White.Width := 80;
  White.Hint := 'w';
  a[1] := o1; a[2] := o2; a[3] := o3;
  a[4] := o4; a[5] := o5;
```

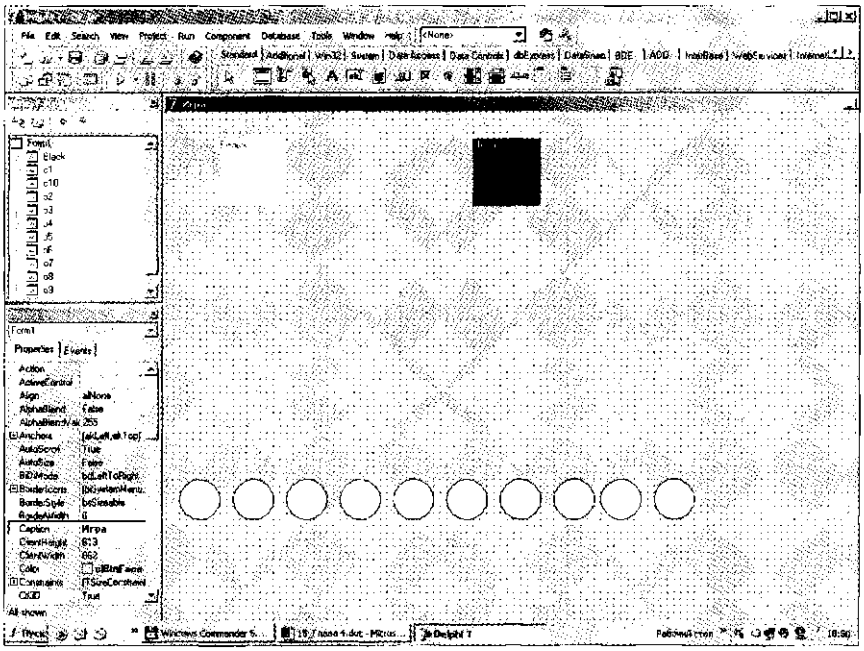


Рис. 4.12. Форма для создания интерфейса приложения «Игра»

```
    a[6] := o6; a[7] := o7; a[8] := o8;
    a[9] := o9; a[10] := o10;
    randomize;
    n := random(9) + 2;
    {Скрываем лишние шарики}
    For i := n + 1 To 10 Do
    With a[i] Do Visible := False;
  {Раскрашиваем в белый или черный цвет оставшиеся
  шарики}
    For i := 1 To n Do
    With a[i] Do
    Begin
         If random(2) = 0
         Then  Begin
                    Brush.Color := clBlack;
                    Hint :='b'
               End
         Else  Begin
                    Brush.Color := clwhite;
```

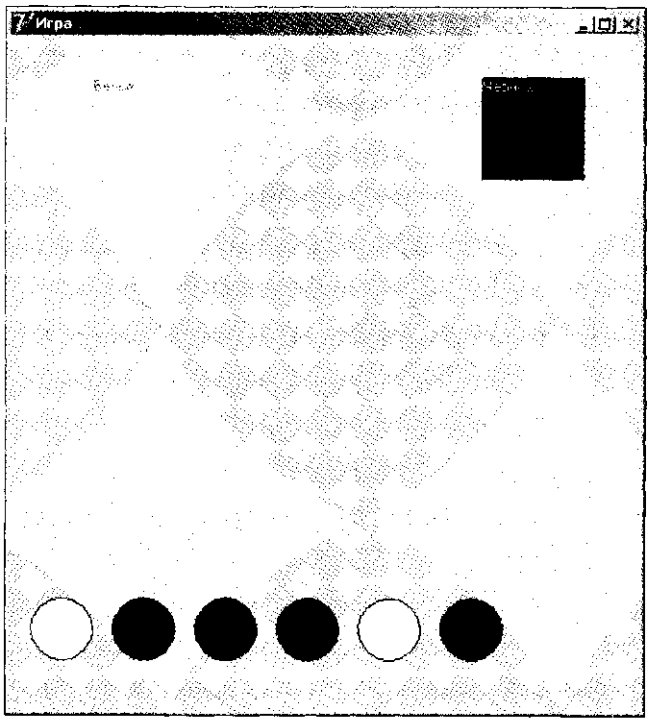


Рис. 4.13. Окно приложения «Игра»

```
                    Hint := 'w'
                End;
    DragMode := dmAutomatic
    End;
End;
```

Здесь *b*, *w* — соответственно счетчики черных и белых шариков, уже опущенных в коробки; Black, White — имена компонентов, т. е. «коробок» для хранения черных и белых шариков.

Установка свойства Hint осуществляется для того, чтобы у шарика, который можно разместить в данной коробке, данное свойство имело такое же значение.

Информацию о каждом из шариков (для удобства дальнейшего оперирования) размещаем в массиве.

Сгенерировав число отображаемых шариков, лишние скрываем, оставшиеся раскрашиваем в случайный цвет (черный или бе-

лый), устанавливаем для шарика определенного цвета соответствующее значение свойства Hint.

Свойство DragMode определяет, как будет выполняться весь комплекс действий, связанных с Drag&Drop. Если DragMode принимает значение dmManual, то все события перетаскивания должны определяться вручную (т.е. программистом по ходу выполнения программы), и начинается перетаскивание только после вызова специальных методов. Если DragMode принимает значение dmAutomatic, то все события перетаскивания определяются автоматически, а перетаскивание начинается сразу после нажатия клавиши мыши пользователем. В нашем случае устанавливается значение dmAutomatic.

Далее необходимо обработать события, связанные с перетаскиванием объектов.

Проверка готовности приемника принять перетаскиваемый объект происходит при обработке события OnDragOver.

Программа проверки следующая:

```
Procedure TForm1.BlackDragOver(Sender, Source :
TObject; X, Y : Integer; State: TDragState; Var
Accept : Boolean);
Begin
  If (Sender Is TLabel) And (Source Is TShape) Then
  Accept := ((Sender As TLabel).Hint = (Source As
TShape).Hint)
End;
```

Параметр Sender указывает компонент, над которым перемещается объект. Параметр Source содержит информацию о компоненте-отправителе. В параметрах *X* и *Y* содержатся координаты указателя мыши, выраженные в пикселах, относительно компонента Sender. Параметр State указывает состояние перемещаемого объекта относительно компонента Sender. Параметр Accept сообщает, готов ли Sender принять перетаскиваемые данные. Если параметр имеет значение True, то Sender готов принять перетаскиваемый объект.

В данном примере обработка указанного события заключается в сравнении значений свойства Hint отправителя и приемника.

Обработка события реализована для объекта Black. В объекте White для соответствующего события сделана ссылка на этот же обработчик события.

При завершении перетаскивания (вне зависимости от того, приняты данные или нет) для перетаскиваемого объекта возника-

ет событие OnDragEnd. Это событие происходит также при отмене перетаскивания, причем оно не является обязательным для выполнения. Операция перетаскивания может быть произведена и без обработки этого события.

Реализуем обработку для первого шарика (для остальных сделана ссылка на этот же обработчик события):

```
Procedure TForm1.o1EndDrag(Sender, Target:
TObject; X, Y: Integer);
Var s: String;
Begin
  If Target <> Nil Then
  Begin
    If (Target As TLabel).Name = 'Black'
    Then Begin b := b + 1; str(b, s);
                (Target As TLabel).Caption :=
                'Черных' + #10 + #13 + s
         End
    Else Begin w := w + 1; str(w, s);
                (Target As TLabel).Caption :=
                'Белых' + #10 + #13 + s
         End;
    (Sender As TShape).Visible := False;
  End;
End;
```

При этом увеличим соответствующий счетчик принятых шариков на единицу и скроем сброшенный в коробку шарик.

Полный вариант программного модуля следующий:

```
Unit Ex_3;
Interface
Uses Windows, Messages, SysUtils, Variants,
Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, ExtCtrls;
Type
  TForm1 = Class(TForm)
       White : TLabel;
       Black : TLabel;
       o1 : TShape;
       o2 : TShape;
       o3 : TShape;
       o4 : TShape;
```

```
        o5 : TShape;
        o6 : TShape;
        o7 : TShape;
        o8 : TShape;
        o9 : TShape;
        o10 : TShape;
        Procedure FormCreate(Sender : TObject);
        Procedure BlackDragOver(Sender, Source :
TObject; X, Y : Integer; State : TDragState; Var
Accept : Boolean);
        Procedure o1EndDrag(Sender, Target :
TObject; X, Y : Integer);
  End;
Var Form1 : TForm1; n : Integer;
    a : Array[1..10] Of TShape;
    b, w: byte;
Implementation
   {$R *.dfm}
Procedure TForm1.FormCreate(Sender: TObject);
Var i : Integer;
Begin
  b := 0;
  w := 0;
  Black.Height := 80;
  Black.Width := 80;
  Black.Hint := 'b';
  White.Height := 80;
  White.Width := 80;
  White.Hint := 'w';
  a[1] := o1; a[2] := o2;
  a[3] := o3; a[4] := o4;
  a[5] := o5; a[6] := o6;
  a[7] := o7; a[8] := o8;
  a[9] := o9; a[10] := o10;
  randomize;
  n := random(9) + 2;
  {Скрываем лишние шарики}
  For i := n + 1 To 10 Do
    With a[i] Do Visible := False;
    {Раскрашиваем в белый или черный оставшиеся
    шарики}
  For i := 1 To n Do
    With a[i] Do
```

```
    Begin
      If random(2) = 0
      Then
      Begin Brush.Color := clBlack; Hint := 'b' End
      Else
      Begin Brush.Color := clwhite; Hint := 'w' End;
      DragMode := dmAutomatic
    End;
End;
Procedure TForm1.BlackDragOver(Sender, Source :
TObject; X, Y : Integer; State : TDragState;
Var Accept: Boolean);
Begin
  If (Sender Is TLabel) And (Source Is TShape)
  Then
    Accept := ((Sender As TLabel).Hint = (Source
As TShape).Hint)
End;
Procedure TForm1.o1EndDrag(Sender, Target :
TObject; X, Y : Integer);
Var s : String;
Begin
  If Target <> Nil Then
  Begin
    If (Target As TLabel).Name = 'Black'
    Then Begin b := b + 1; str(b, s);
                (Target As TLabel).Caption:=
                'Черных' + #10 + #13 + s
         End
    Else Begin w := w + 1; str(w, s);
                (Target As TLabel).Caption :=
                'Белых' + #10 + #13 + s
      End;
    (Sender As TShape).Visible := False;
  End;
End;
End.
```

УПРАЖНЕНИЯ

1. Продолжить программу из примера 4.4, реализовав по аналогии процедуры для других цифровых кнопок и кнопок со знаками операций, и получить работающий калькулятор.

2. Дополнить форму к примеру 4.5 (см. рис. 4.12) кнопкой [Закрыть], которая появляется после того, как все шарики помещены в коробки. Обработать нажатие кнопки, по которому закрывается приложение.

4.8. ИЕРАРХИЯ КЛАССОВ

Терминология, касающаяся объектно-ориентированного программирования, принятая в Delphi, несколько отличатся от терминологии в Турбо Паскале. Рассмотрим эту терминологию, а также способы реализации объектного подхода к программированию и основные его принципы, принятые в Delphi.

Классы в Delphi. Объявление класса, свойств и методов. То, что в Турбо Паскале именовалось объектом, в Delphi принято называть классом. *Классы* в Delphi — это специальные типы, которые содержат следующие элементы: поля, методы и свойства. Как и любой другой тип, класс служит лишь образцом для создания конкретных экземпляров реализации, которые называются *объектами*. Данная терминология употребляется в C++, откуда и была заимствована.

Важным отличием классов от других типов является то, что их объекты всегда распределяются в неупорядоченной структуре (куче), поэтому объект «Переменная» фактически представляет собой лишь указатель на динамическую область памяти. Однако в отличие от других указателей при ссылке на содержимое объекта запрещается использовать операцию разыменования, т.е. символ «^» за именем объекта.

Все классы, используемые в Delphi (как библиотечные, так и разрабатываемые программистом), наследуются от класса TObject. При этом указывать данный класс при объявлении как «родительский» необязательно.

Формат описания нового класса следующий:

```
Type <идентификатор типа класса> = Class
  Private <скрытые элементы класса>;
  Protected <защищенные элементы класса>;
  Public <общедоступные элементы класса>;
  Published <опубликованные элементы класса>;
End;
```

Не все указанные в данном шаблоне секции могут присутствовать при объявлении класса. Поясним их смысл.

Секция Private содержит внутренние элементы, обращение к которым возможно только в пределах модуля, содержащего объявление класса.

Секция Protected содержит защищенные элементы, которые доступны в пределах модуля, включающего в себя объявление класса, а также внутри «потомков» класса.

Секция Public содержит общедоступные элементы, к которым возможно обращение из любой части программы.

Наконец, секция Published содержит опубликованные элементы, доступ к которым осуществляется так же, как и элементам секции Public.

«Потомки» класса могут менять область доступности всех элементов «родительского» класса, за исключением объявленных в секции Private.

Если при объявлении элементов класса не указывать, к какой секции относится объявление, то доступ к данным элементам будет осуществляться как к объявленным в секции Public. В Delphi разрешается сколько угодно раз объявлять любую секцию, причем порядок следования секций не имеет значения и любая секция может быть пустой.

Приведем пример объявления класса. Опишем класс «Обыкновенная дробь» с методами «НОД числителя и знаменателя», «Сокращение», «Натуральная степень» (см. пример 4.1):

```
Type Natur = 1..32767;
     Frac = Record P : Integer; Q : Natur End;
     Drob = Class
       Procedure Vvod; {Ввод дроби}
       Procedure Print; {Вывод дроби}
       Procedure NOD(Var C : Natur);
       Procedure Sokr;
       Procedure Stepen(N : Natur; Var C : Frac);
       Property A : Frac Read Print Write Vvod;
     End;
```

Здесь объединение полей, методов и свойств в единое целое, как и в Турбо Паскале, называется инкапсуляцией. Поля могут быть любого типа, в том числе и классами. Инкапсулированные в классе процедуры и функции являются методами. Свойства — это специальный механизм классов, регулирующий доступ к полям. Объявляются свойства с помощью зарезервированных слов Property, Read и Write (слова Read и Write считаются зарезервированными только в контексте объявления свойства). Обычно свой-

ство связано с некоторым полем и указывает те методы класса, которые должны использоваться при записи в это поле или при считывании из него.

Реализация методов, как и в Турбо Паскале, осуществляется вне объявления класса, при этом по-прежнему требуется указывать метод какого класса реализуется. Например, при реализации метода Vvod из примера 4.1 заголовок будет иметь следующий вид:

```
Procedure Drob.Vvod;
```

Наследование. Любой класс может быть образован от другого класса. Для этого при его объявлении указывается имя класса-«родителя»:

```
<дочерний класс> = Class (<родительский класс>)
```

Образованный класс автоматически наследует поля, методы и свойства своего «родителя» и может добавлять новые. Принцип наследования обеспечивает поэтапное создание сложных классов и разработку собственных библиотек классов.

Все классы Delphi образованы от одного «родителя» — класса TObject, который не имеет полей и свойств, но включает в себя методы самого общего назначения, обеспечивающие весь жизненный цикл любых объектов — от создания до уничтожения.

Принцип наследования приводит к созданию ветвящегося дерева классов, постепенно разрастающегося при перемещении от класса TObject к его «потомкам». Каждый «потомок» дополняет возможности своего «родителя» новыми и передает их своим «потомкам».

В Delphi не реализовано множественное наследование, т. е. «дочерний» класс может иметь только одного «родителя».

В качестве примера приведем рассмотренный ранее класс «Вычислитель» (см. пример 4.2):

```
Type BaseType = Double;
     Vichislitel = Class
     A, B, C : BaseType;
     Procedure Init; {Ввод или инициализация полей}
     Procedure Slozh;
     Procedure Vich;
     Procedure Umn;
     Procedure Delen;
End;
```

```
    NovijVichislitel = Class(Vichislitel)
    N : Integer;
    Procedure Stepen;
    Procedure Koren;
End;
```

Полиморфизм, перегрузка методов. В Delphi определение полиморфизма аналогично определению в Турбо Паскале: полиморфизм — это свойство классов выполнять одинаковые по смыслу действия разными способами. Изменив алгоритм того или иного метода «потомков» класса, придадим им отсутствующие у «родителя» специфические свойства.

Для изменения метода необходимо перекрыть его у «потомка», т.е. объявить у «потомка» одноименный метод и реализовать в нем необходимые действия. В результате объект «родитель» и объект «потомок» будут иметь два одноименных метода с разной алгоритмической основой, которые, следовательно, придают объектам разные свойства.

В Delphi полиморфизм достигается не только с помощью механизма наследования и перекрытия методов «родителя», но и их виртуализацией, позволяющей «родительским» методам обращаться к методам своих «потомков».

Механизм перекрытия «родительского» метода одноименным методом «потомка» приводит к тому, что последний «не видит» перекрытый «родительский» метод и может обращаться к нему лишь с помощью зарезервированного слова Inherited. В Delphi введено зарезервированное слово Overload (перезагрузить), с помощью которого становятся видны одноименные методы как «родителя», так и «потомка».

Понятие объектно-ориентированного проектирования. Модели разрабатываемого программного обеспечения при объектном подходе основаны на предметах и явлениях реального мира, а также на описании требуемого поведения разрабатываемого программного обеспечения, т.е. его функциональности, которая связывается с состояниями элементов (объектов) конкретной предметной области.

Объектный подход к разработке программного обеспечения основан на использовании *объектной декомпозиции*, т.е. представлении разрабатываемого программного обеспечения в виде совокупности объектов, в процессе взаимодействия которых через передачу сообщений и происходит выполнение требуемых функций.

Однако при объектном подходе, как и при структурном, сразу можно выполнить декомпозицию только очень простого программного обеспечения. Поэтому сначала для объектно-ориентированного программирования были разработаны соответствующие методы анализа и проектирования программного обеспечения, использующие различные модели и нотации.

В 1994 г. был разработан язык UML (Unified Modeling Language — унифицированный язык моделирования), который в настоящее время фактически признан стандартным средством описания проектов, создаваемых с использованием объектно-ориентированного подхода. (Подробное описание этого языка смотрите в специальной литературе.)

УПРАЖНЕНИЯ

1. Описать следующие классы с указанием их методов:
   а) «Телефонный звонок» — номер телефона, дата разговора, продолжительность, код города и т.д.;
   б) «Поездка» — номер поезда, пункт назначения, дни следования, время прибытия, время стоянки и т.д.;
   в) «Кинотеатр» — название кинофильма, сеанс, стоимость билета, количество зрителей и т.д.

Реализовать объявленные в классах методы. Описать объекты-«потомки» для указанных классов. Выполнить отладку и тестирование приложения.

2. Реализовать в виде класса пример 4.3. Дополнить набор классов классом «Прямоугольник», а также другими выпуклыми четырехугольниками, не являющимися параллелограммами (трапеция и т.д.), и реализовать соответственно их методы.

3. Реализовать в среде Delphi упражнение 6 из подразд. 4.2.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем состоит отличие парадигмы объектно-ориентированного программирования от процедурного программирования?
2. Как определяется понятие объекта в Object Pascal?
3. Что такое инкапсуляция, наследование, полиморфизм?
4. Когда и для каких целей была разработана система программирования Delphi?
5. В чем заключается суть визуальной технологии программирования?
6. Каковы основные составляющие интерфейса среды Delphi и их назначение?

7. Какие основные компоненты (объекты) интерфейса приложений, создаваемых в Delphi, вам известны?
8. Что такое событийно-управляемое программирование?
9. Что такое «событие» применительно к различным объектам графического интерфейса?
10. Каким образом определяется реакция компьютера на событие?
11. Какие этапы разработки приложения в Delphi вы знаете? Охарактеризуйте их.

Приложения

Приложение 1

Турбо Паскаль. Модуль CRT

Таблица П1.1. Константы режимов работы

Имя константы	Номер режима	Режим
BW40	0	Черно-белый, 40 символов, 25 строк
CO40	1	Цветной, 40 × 25
BW80	2	Черно-белый, 80 × 25
CO80	3	Цветной, 80 × 25
Mono	7	Монохромный, 80 × 25, для монохромных дисплеев

Таблица П1.2. Константы цветов

Имя константы	Номер цвета	Цвет
Black	0	Черный
Blue	1	Темно-синий
Green	2	Темно-зеленый
Cyan	3	Бирюзовый
Red	4	Красный
Magenta	5	Фиолетовый
Brown	6	Коричневый
LightGray	7	Светло-серый
DarkGray	8	Темно-серый
LightBlue	9	Синий
LightGreen	10	Светло-зеленый
LightCyan	11	Светло-бирюзовый

Окончание табл. П1.2

Имя константы	Номер цвета	Цвет
LightRed	12	Розовый
LightMagenta	13	Малиновый
Yellow	14	Желтый
White	15	Белый
Blink	128	Мерцание символа

Таблица П1.3. Процедуры и функции управления режимами

Интерфейс	Назначение
Установка режимов и окон	
Procedure AssignCrt (File : Text);	Связывает окно дисплея с текстовым файлом, что позволяет ускорить вывод на экран
Procedure ClrScr;	Очищает экран и помещает курсор в верхний левый угол
Procedure TextMode (Mode : Integer); Mode — номер текстового режима или соответствующая константа	Выбирает текстовый режим
Procedure Window (*x1*, *y1*, *x2*, *y2* : Byte); (*x1*, *y1*) и (*x2*, *y2*) — координаты верхнего левого и нижнего правого углов окна	Определяет окно вывода в текстовом режиме
Управление цветом текста и фона	
Procedure HighVideo;	Устанавливает высокую яркость выводимых символов
Procedure LowVideo;	Устанавливает низкую яркость выводимых символов
Procedure NormVideo;	Возвращает цвет символов и фона, свойственный данному графическому режиму по умолчанию
Procedure TextBackground (Color : Byte); Color — код цвета или соответствующая константа	Выбирает цвета символов

Окончание табл. П1.3

Интерфейс	Назначение
Управление выводом текста	
Procedure ClrEol;	Стирает все символы от текущей позиции курсора до конца строки
Procedure DelLine;	Удаляет линию, в которой находится курсор
Procedure InsLine;	Вставляет новую строку текста перед строкой, где находится курсор
Работа с клавиатурой	
Function KeyPressed : Boolean; Значение True, если нажата любая клавиша, и False, если не нажата	Определяет, была ли нажата клавиша на клавиатуре
Function ReadKey : Char; Значение функции — код символа клавиши, нажатой на клавиатуре	Считывает символ из буфера клавиатуры
Управление курсором	
Procedure GotoXY (X,Y : Integer); *X*, *Y* — координаты курсора	Перемещает курсор в указанные координаты окна вывода
Function WhereX : Integer; Значение функции — координата X курсора	Возвращает текущую координату *X* курсора
Function WhereY : Integer; Значение функции — координата *Y* курсора	Возвращает текущую координату *Y* курсора
Управление звуком	
Procedure NoSound;	Включает динамик
Procedure Sound (Hz : Word); Hz — частота звука в герцах	Включает звук динамика с заданной тональной частотой
Управление временем	
Procedure Delay (Ms : Word); *Ms* — значение задержки в миллисекундах	Задерживает исполнение программы на заданное число миллисекунд

Турбо Паскаль. Модуль GRAPH

Таблица П2.1. Коды драйверов графических устройств

Имя	Значение	Назначение
Detect	0	Автоматический выбор драйвера
CGA	1	
MCGA	2	
EGA	3	
EGA64	4	
EGAMono	5	
IBM8514	6	
HercMono	7	
ATT400	8	
VGA	9	
PC3270	10	
CurrentDriver	–128	Текущий драйвер

Таблица П2.2. Константы графических режимов

Имя	Значение	Размер поля	Палитра	Число страниц
ATT400C0	0	320 × 200	C0	1
ATT400C1	1	320 × 200	C1	1
ATT400C2	2	320 × 200	C2	1
ATT400C3	3	320 × 200	C3	1
ATT400Med	4	640 × 200	2 цвета	1
ATT400Hi	5	640 × 400	2 цвета	1
CGAC0	0	320 × 200	C0	1
CGAC1	1	320 × 200	C1	1
CGAC2	2	320 × 200	C2	1

Окончание табл. П2.2

Имя	Значение	Размер поля	Палитра	Число страниц
CGAC3	3	320 × 200	C3	1
CGACHi	4	640 × 200	2 цвета	1
EGALo	0	640 × 200	16 цветов	4
EGAHi	1	640 × 350	16 цветов	2
EGA64Lo	0	640 × 200	16 цветов	1
EGA64Hi	1	640 × 350	4 цвета	1
EGAMonoHi	0	640 × 350	2 цвета	1 или 2
HercMonoHi	0	720 × 348	2 цвета	2
IBM8514Lo	0	640 × 480	256 цветов	1
IBM8514Hi	0	1024 × 768	256 цветов	1
MCGAC0	0	320 × 200	C0	1
MCGAC1	1	320 × 200	C1	1
MCGAC2	2	320 × 200	C2	1
MCGAC3	3	320 × 200	C3	1
MCGAMed	4	640 × 200	2 цвета	1
MCGAHi	5	640 × 480	2 цвета	1
PC3270Hi	0	720 × 350	2 цвета	1
VGALo	0	640 × 200	16 цветов	4
VGAMed	1	640 × 350	16 цветов	2
VGAHi	2	640 × 480	16 цветов	1

Примечание. Палитра C0 включает в себя цвета светло-зеленый, розовый и желтый, палитра C1 — светло-голубой, светло-фиолетовый и белый, палитра C2 — зеленый, красный и коричневый, палитра C3 — голубой, фиолетовый и светло-серый.

Таблица П2.3. Константы цветов

Имя константы	Номер цвета	Цвет
Black	0	Черный
Blue	1	Темно-синий

Окончание табл. П2.3

Имя константы	Номер цвета	Цвет
Green	2	Темно-зеленый
Cyan	3	Бирюзовый
Red	4	Красный
Magenta	5	Фиолетовый
Brown	6	Коричневый
LightGray	7	Светло-серый
DarkGray	8	Темно-серый
LightBlue	9	Синий
LightGreen	10	Светло-зеленый
LightCyan	11	Светло-бирюзовый
LightRed	12	Розовый
LightMagenta	13	Малиновый
Yellow	14	Желтый
White	15	Белый

Таблица П2.4. Коды линий

Имя	Значение	Назначение
Коды типов линий (для процедуры SetLineStyle)		
SolidLn	0	Сплошная
DottedLn	1	Пунктирная
CenterLn	2	Штрихпунктирная
DashedLn	3	Штриховая
UserBitLn	4	Заданная пользователем
Коды толщины линии		
NormWidth	1	Нормальная
ThickWidth	3	Толстая

Таблица П2.5. Константы типа заполнения (для процедуры SetFillStyle)

Имя константы	Значение	Назначение
EmptyFill	0	Заполнение цветом фона
SolidFill	1	Однородное заполнение цветом
LineFill	2	Заполнение вида —
LtSlashFill	3	Заполнение вида ///
SlashFill	4	Заполнение вида /// толстыми линиями
BkSlashFill	5	Заполнение вида \ толстыми линиями
LtBkSlashFill	6	Заполнение вида \
HatchFill	7	Заполнение клеткой
XHatchFill	8	Заполнение косой клеткой
InterleaveFill	9	Заполнение частой клеткой
WideDotFill	10	Заполнение редкими точками
CloseDotFill	11	Заполнение частыми точками
UserFill	12	Определяется пользователем

Таблица П2.6. Процедуры и функции

Интерфейс	Назначение
Управление графическим режимом	
Procedure CloseGraph;	Закрывает графический режим
Procedure DetectGraph (Var GrDriver, GrMode: Integer); GrDriver — код драйвера, GrMode — код графического режима	Определяет рекомендуемые к применению для данного компьютера графические драйвер и режим
Function GetDriverName : String; Значение функции — имя используемого драйвера	Определяет имя файла с используемым графическим драйвером
Function GetGraphMode : Integer; Значение функции — код графического режима	Определяет код используемого графического режима

Продолжение табл. П2.6

Интерфейс	Назначение
Function GetMaxName : Integer; Значение функции — имя используемого графического режима	Определяет имя используемого графического режима
Function GetMaxMode : Integer; Значение функции — максимальное значение кода режима	Определяет максимальное значение кода графического режима для используемого драйвера
Procedure GetModeRange (GrDriver : Integer; Var LoMode, HiMode : Integer); GrDriver — код графического режима; LoMode, HiMode — наименьшее и наибольшее значения кода графического режима для данного драйвера	Определяет минимальное и максимальное значения кода графического режима для указанного при обращении драйвера
Procedure GraphDefaults;	Устанавливает графический указатель в начало координат и переустанавливает графическую систему
Function GraphErrorMsg (ErrorCode : Integer) : String; ErrorCode — код графической ошибки. Значение функции — текстовое сообщение о характере ошибки	Дает строку — сообщение об ошибке графического режима по коду ошибки
Function GraphResult : Integer; Значение функции — код ошибки	Определяет, произошла ли ошибка при исполнении процедур модуля
Procedure InitGraph (var GrDriver, GrMode : Integer; PathToDriver : String); GrDriver — код драйвера, GrMode — код графического режима; PathToDriver — путь к файлу используемого драйвера	Устанавливает заданный графический режим
Function InstallUserDriver (Name : String; AutoDetectPtr : Pointer) : Integer; Name — имя файла графического драйвера, AutoDetectPtr — указатель на процедуру, определяющую успешность запуска драйвера. Значение	Инсталлирует пользовательский драйвер графического режима

Продолжение табл. П2.6

Интерфейс	Назначение
функции — цифровой код установленного драйвера	
Procedure RegisterBGIDriver (Driver : Pointer) : Integer; Driver — указатель на драйвер	Регистрирует драйвер графической системы
Procedure RestoreCrtMode;	Закрывает графический режим и восстанавливает текстовый режим, установленный ранее
Procedure SetGraphMode(GrMode : Integer);GrMode — код графического режима	Устанавливает другой графический режим без изменения драйвера
Управление экраном и окнами	
Procedure ClearDevicc;	Очищает экран, сбрасывает все графические установки к значениям по умолчанию, устанавливает графический указатель в положение (0, 0)
Procedure ClearViewPort;	Очищает экран, устанавливая фон, заданный в SetBkColor
Function GetMaxX : Integer; Значение функции — максимальная координата *X*	Определяют максимальные координаты экрана в данном графическом режиме
Function GetMaxY : Integer; Значение функции — максимальная координата *Y*	
Procedure GetAspectRatio (Var Xasp, Yasp: Word); Xasp, Yasp — коэффициенты по осям *X* и *Y*	Определяет коэффициенты, характеризующие неодинаковость линейного расстояния между пикселами по осям *X* и *Y*
Function GetBkColor : Word; Значение функции — код цвета фона	Определяет установленный цвет фона
Экран и окна	
Procedure GetViewSettings (Var ViewPort : ViewPortType); ViewPort — параметры текущего окна	Запрашивает текущие параметры окна и отсечения

Продолжение табл. П2.6

Интерфейс	Назначение
Procedure SetBkColor (Colorv: Word); Значение функции — код цвета фона	Устанавливает цвет фона
Procedure SetActivePage (Page : Word); Page — номер активной страницы	Устанавливает активную графическую страницу
Procedure SetAspectRatio (Var Xasp, Yasp : Word); Xasp, Yasp — коэффициенты по осям *X* и *Y*	Изменяет масштабный коэффициент отношения сторон экрана
Procedure SetVisualPage (Page : Word); Page — номер страницы	Устанавливает номер видимой графической страницы
Procedure SetViewPort (*x*1, *y*1, *x*2, *y*2 : Integer; Clip : Boolean); (*x*1, *y*1) и (*x*2, *y*2) — координаты противоположных углов окна; Clip — определяет, отсекать ли изображение за пределами окна или нет	Устанавливает визуальный порт (окно) для вывода графического изображения
Function GetColor : Word; Значение функции — код цвета	Возвращает текущий цвет (установленный SetColor)
Procedure GetDefaultPalette (Var Palette : PaletteType); Palette — палитра (массив типа PaletteType)	Определяет, какая палитра действует в режиме по умолчанию
Function GetMaxColor : Word; Значение функции — максимальный код цвета	Определяет максимальный код цвета в установленном режиме
Procedure GetPalette (Var Palette : PaletteType); Palette — палитра (массив типа PaletteType)	Определяет, какая палитра установлена
Function GetPaletteSize : Word; Значение функции — число цветов в палитре	Возвращает размер таблицы палитры
Procedure SetAllPalette (Var Palette : PaletteType); Palette — палитра (массив типа PaletteType)	Устанавливает все цвета палитры
Procedure SetColor (Color : Word); Color — код цвета	Устанавливает цвет для линий

Продолжение табл. П2.6

Интерфейс	Назначение
Procedure SetPalette(ColorNum, Color : Word); ColorNum — код заменяемого цвета; Color — код нового цвета	Устанавливает один новый цвет в палитре
Procedure SetRGBPalette (ColorNum, RedValue, GreenValue, BlueValue : Word); ColorNum — код устанавливаемого в палитре цвета; RedValue, GreenValue, BlueValue — интенсивность красного, зеленого и синего в этом цвете	Устанавливает в палитру цвет, заданный тремя компонентами: красным, зеленым и синим
Управление указателем	
Function GetX : Integer; GetX — координата *X* указателя	Возвращает координату *X* текущего указателя
Function GetY : Integer; GetY — координата *Y* указателя	Возвращает координату *Y* текущего указателя
Procedure MoveTo (X : Integer; Y : Integer); (*X*, *Y*) — координаты точки экрана	Перемещает указатель в точку, заданную координатами *X*, *Y*
Procedure MoveRel (dx : Integer; dy : Integer); dx, dy — вектор приращений координат точки экрана	Перемещает указатель в точку, координаты которой отличаются от текущих координат на значения *dx*, *dy*
Шаблоны; закраска областей	
Procedure FloodFill (X, Y : Integer; ColorBorder : Word); (*X*, *Y*) — координаты точки, вокруг которой идет закраска; ColorBorder — цвет, обозначающий границы области закраски	Закрашивает произвольную область по заданным шаблону и цвету вокруг заданной точки до границ, обозначенных определенным цветом
Procedure GetFillPattern (Var FillPattern : FillPatternType); FillPattern — массив типа FillPatternType, содержащий информацию о шаблоне	Определяет установленный тип шаблона
Procedure GetFillSettings (Var FillInfo : FillSettingsType);	То же

Продолжение табл. П2.6

Интерфейс	Назначение
FillInfo — запись типа FillSettingsType, содержащая информацию о шаблоне	
Procedure GetLineSettings (Var LineInfo : LineSettingsType); LineInfo — запись типа LineSettingsType, содержащая информацию о типе линии	Определяет установленный тип линии
Procedure GetGraphBufSize (Size: Word); Size — размер буфера	Изменяет размер буфера для функций заполнения
Procedure SetFillPattern (Pattern : FillPatternType; Color : Word); Pattern — массив типа FillPatternType, содержащий информацию о шаблоне; Color — цвет раскраски	Устанавливает шаблон и цвет раскраски
Procedure SetFillStyle (Pattern, Color : Word); Pattern — код стандартного шаблона; Color — цвет раскраски	Устанавливает один из стандартных шаблонов и цвет раскраски
Procedure SetLineStyle (LineStyle, Pattern, Thickness : Word); LineStyle — код стиля линии; Pattern — собственный шаблон линии; Thickness — толщина линии	Устанавливает стиль линии как один из стандартных или по собственному шаблону
Изображение геометрических фигур	
Procedure Arc (X, Y : Integer; StartAngle, EndAngle, R : Word); (*X*, *Y*) — координаты центра окружности; StartAngle, EndAngle — начальный и конечный углы дуги окружности в градусах (0,359); *R* — радиус окружности в пикселах в направлении *X*	Изображает дугу окружности. Процедура учитывает неодинаковость масштаба по осям
Procedure Bar (x1, y1, x2, y2 : Integer); (*x1*, *y1*) и (*x2*, *y2*) — координаты левого верхнего и правого нижнего углов прямоугольника	Изображает окрашенный прямоугольник без контура

Продолжение табл. П2.6

Интерфейс	Назначение
Procedure Bar3D (x1, y1, x2, y2 : Integer; Depth : Word; Top : boolean); (*x*1, *y*1) и (x2, *y*2) — координаты левого верхнего и правого нижнего углов параллелепипеда, Depth — его глубина; Top — определяет, изображать ли верхнюю грань фигуры (true — изображать)	Изображает трехмерный прямоугольный параллелепипед с заполнением передней грани. Верхняя грань может изображаться или не изображаться
Procedure Circle (x, y : Integer; R : Word); (*x*, *y*) — координаты центра окружности; *R* — радиус окружности в пикселах в направлении *X*	Изображает окружность. Процедура учитывает неодинаковость масштаба по осям
Procedure DrawPoly (NumPoints : Word; Var PolyPoints); NumPoints — число точек, задающих ломаную линию; PolyPoints — массив точек (элементов типа Point), задающих ломаную линию	Изображает ломаную линию, проходящую через данный массив точек
Procedure Ellipse (X, Y : Integer; StartAngle, EndAngle, Rx, Ry : Word); (*X*, *Y*) — координаты центра эллипса; StartAngle, EndAngle — начальный и конечный углы дуги эллипса в градусах (0,359); *Rx*, *Ry* — полуоси эллипса в пикселах в направлениях *X* и *Y*	Изображает дугу эллипса, полуоси которого в направлениях *X* и *Y* заданны в пикселах
Procedure GetArcCoords (Var ArcCoords : ArcCoordsType); ArcCoordsType — запись, содержащая координаты дуги	Возвращает координаты дуги, изображенной процедурами Arc и Ellipse
Procedure FillEllipse (X, Y : Integer; Rx, Ry : Word); (*X*, *Y*) — координаты центра эллипса; *Rx*, *Ry* — полуоси эллипса в пикселах в направлениях *X* и *Y*	Чертит закрашенный эллипс, полуоси которого в направлениях *X*, *Y* заданы в пикселах
Procedure FillPoly (NumPoints : Word; Var PolyPoints); NumPoints — число точек, задающих ломаную линию;	Изображает закрашенный многоугольник, заданный массивом вершин

Продолжение табл. П2.6

Интерфейс	Назначение
PolyPoints — массив точек (элементов типа Point), задающих ломаную линию	
Function GetPixel (x, y : Integer) : Word; (*x*, *y*) — координаты точки, результат — цвет точки	Возвращает цвет заданной точки
Procedure Line (x1, y1, x2, y2 : Integer); (*x1*, *y1*), (*x2*, *y2*) — координаты конечных точек отрезка прямой	Чертит отрезок прямой по двум заданным конечным точкам
Procedure LineRel (dx, dy : Integer); (*dx*, *dy*) — смещение конечной точки относительно начального положения указателя	Чертит отрезок прямой от положения указателя до точки, смещенной относительно указателя на заданное значение
Procedure LineTo (x, y : Integer); (*x*, *y*) — координаты точки, в которую проводится отрезок	Чертит отрезок прямой от точки положения указателя до заданной точки
Procedure Rectangle (x1, y1, x2, y2 : Integer); (*x1*, *y1*), (*x2*, *y2*) — координаты противоположных углов прямоугольника	Чертит контур прямоугольника
Procedure Sector (X, Y : Integer; StartAngle, EndAngle, Rx, Ry : Word); (*X*, *Y*) — координаты центра эллипса; StartAngle, EndAngle — начальный и конечный углы дуги эллипса в градусах (0,359); *Rx*, *Ry* — полуоси эллипса в пикселах в направлениях *X* и *Y*	Изображает закрашенный сектор эллипса, ограниченный углами от StartAngle до EndAngle. Заполняется сектор от минимального значения углов до максимального независимо от их следования
Procedure PieSlice (X, Y : Integer; StartAngle, EndAngle, Rx : word); (*X*, *Y*) — координаты центра эллипса; StartAngle, EndAngle — начальный и конечный углы дуги эллипса в градусах (0, 359); *Rx* — радиус окружности в пикселах в направлении *X*	Изображает закрашенный сектор круга, ограниченный углами от StartAngle до EndAngle, радиус круга, заданный в пикселах в направлении *X*, и учитывает масштаб изображения по осям. Заполняется сектор от минимального значения углов до максимального независимо от их следования

Продолжение табл. П2.6

Интерфейс	Назначение
Procedure PutPixel (X, Y : Integer; Color : word); (*X*, *Y*) — координаты точки; Color — цвет точки	Окрашивает точку экрана в заданный цвет
Вывод текстов	
Procedure GetTextSettings (Var TextInfo : TextSettingsType); TextInfo — запись, содержащая действующие установки вывода текста	Определяет установки вывода текста
Procedure OutText (Text : String); Text — выводимая строка текста	Выводит текст на графический экран, начиная с позиции графического указателя, которая смещается на ширину выводимого текста
Procedure OutTextXY (X, Y : Integer; Text : String); (*X*, *Y*) — координаты точки, к которой привязывается выводимый текст; Text — выводимая строка текста	Выводит текст на графический экран относительно заданной точки с учетом используемого типа юстировки
Function InstallUserFont (FontFileName : String): Integer; FontFileName — имя файла, содержащего шрифт. Результат — номер (код) установленного шрифта	Инсталлирует новый шрифт
Procedure RegisterBGIFont (Font : Pointer); Font — указатель на шрифт	Регистрирует шрифты для графической системы
Procedure SetTextJustify (Horiz, Vert : Word); Horiz, Vert — коды привязки текста по горизонтали и вертикали	Устанавливает тип привязки текста к точке вывода по горизонтали и вертикали
Procedure SetTextStyle (Font, Direction, CharSize : Word); Font — тип используемого шрифта; Direction — направление вывода надписи; CharSize — размер символов	Устанавливает стиль выводимого текста

Окончание табл. П2.6

Интерфейс	Назначение
Procedure SetUserCharSize (MultX, DivX, MultY, DivY : Word); MultX, DivX, MultY, DivY — коэффициенты умножения (Mult) и деления (Div) по осям *X* и *Y* соответственно	Выполняет масштабирование шрифтов с произвольным дробным масштабом
Function TextHeight (Text : String) : Integer; Text — выводимая строка текста; Результат — высота текста в пикселах	Определяет высоту текстовой строки при заданных установках стиля
Function TextWidth (Text : String) : Integer; Text — выводимая строка текста; Результат — длина текста в пикселах	Определяет длину текстовой строки при заданных установках стиля
Копирование части экрана	
Procedure GetImage (x1, y1, x2, y2 : Integer; Var BitMap: Pointer); (*x*1, *y*1) и (*x*2, *y*2) — координаты противоположных углов прямоугольника, ограничивающего копируемую область экрана; BitMap — указатель области памяти, отведенной для хранения данного изображения	Копирует в ОЗУ прямоугольную область экрана
Function ImageSize (x1, y1, x2, y2 : Integer) : Word; (*x*1, *y*1) и (*x*2, *y*2) — координаты противоположных углов прямоугольника, ограничивающего область экрана. Результат — объем информации в байтах	Определяет размер в байтах памяти, необходимой для хранения прямоугольной области экрана
Procedure PutImage (x, y : Integer; Var BitMap : Pointer; BitBlt : Word); (*x*, *y*) — координаты левого верхнего угла для выводимого изображения; BitMap — указатель области памяти; BitBlt — код логической операции	Выводит изображение из области памяти ЭВМ в указанную область экрана с заданной логической операцией наложения нового изображения на старое

DELPHI. Некоторые подпрогаммы

Таблица П3.1. Подпрограммы для работы с датой и временем

Интерфейс	Назначение
Function Date : TDateTime;	Возвращает текущую дату
Function DateToStr (D : TDateTime) : String;	Преобразует дату в строку символов
Function DateTimeToStr (D : TDateTime) : String;	Преобразует дату и время в строку символов
Function FormatDateTime (Format : String; Value : TDateTime) : String;	Преобразует дату и время из параметра Value в строку символов в соответствии со спецификаторами параметра Format
Function Now : TDateTime;	Возвращает текущие дату и время
Function Time : TDateTime;	Возвращает текущее время
Function TimeToStr (T : TDateTime) : String;	Преобразует время в строку

Таблица П3.2. Подпрограммы для работы со строками

Интерфейс	Назначение
Function AnsiLowerCase (const S : String) : String;	Возвращает исходную строку *S*, в которой все прописные буквы заменены строчными в соответствии с национальной кодировкой Windows (т. е. с учетом кириллицы)
Function AnsiUpperCase (const S : String) : String;	Возвращает исходную строку *S*, в которой все строчные буквы заменены прописными в соответствии с национальной кодировкой Windows
Function Concat (S1 [, S2,..., SN] : String) : String;	Возвращает строку, представляющую собой сцепление строкпараметров *S1*, *S2*,..., *SN*

Продолжение табл. П3.2

Интерфейс	Назначение
Function Copy (S : String; N, K : Integer) : String;	Из строки *S* копирует *K* символов, начиная с символа с номером *N*
Procedure Delete (Var S : String; N, K : Integer);	Удаляет *K* символов из строки *S*, начиная с символа с номером
Procedure Insert (SubSt : String; Var S : String; N : Integer);	Вставляет подстроку SubSt в строку *S*, начиная с символа с номером *N*
Function Length (St : String) : Integer;	Возвращает текущую длину строки *St*
Function LowerCase (const S : String) : String;	Возвращает исходную строку *S*, в которой все латинские прописные буквы заменены строчными
Function Pos (SubSt, S : String) : Integer;	Отыскивает в строке *S* первое вхождение подстроки SubSt и возвращает номер позиции, с которой она начинается. Если подстрока не найдена, возвращается 0
Procedure SetLength (S : String; NewLength : Integer);	Устанавливает новую (меньшую) длину NewLength строки *S*. Если NewLength больше текущей длины строки, обращение к SetLength игнорируется
Function StringOfChar (Ch : Char; K : Integer) : String;	Создает строку, состоящую из *K* раз повторенного символа Ch
Function StringToOleStr (const Source : String) : PWideChar;	Копирует обычную строку в двухбайтную
Function StringToWideChar (const Source : String; Dest : PWideChar; DestSize : Integer) : PWideChar;	Преобразует обычную строку в строку с символами Unicode
Function Uppercase (const S : String) : String;	Возвращает исходную строку *S*, в которой все строчные латинские буквы заменены прописными
Подпрограммы преобразования строк к другим типам	
Function StrToCurr (S : String) : Currency;	Преобразует символы строки *S* в целое число типа Currency. При этом строка не должна содержать ведущих или ведомых пробелов

Продолжение табл. П3.2

Интерфейс	Назначение
Function StrToDate (S : String) : TDateTime;	Преобразует символы строки *S* в дату. При этом строка должна содержать два или три числа, разделенных правильным для Windows разделителем даты. Первое число — правильный день, второе — правильный месяц. Если указано третье число, оно должно задавать год в формате XX или XXXX. Если символы года отсутствуют, дата дополняется текущим годом
Function StrToDateTime (S : String) : TDateTime;	Преобразует символы строки *S* в дату и время. При этом строка должна содержать правильные дату и время, разделенные пробелом
Function StrToFloat (S : String) : Extended;	Преобразует символы строки *S* в вещественное число
Function StrToInt (S : String) : Integer;	Преобразует символы строки *S* в целое число
Function StrToIntDef (S : String; D : Integer) : Integer;	Преобразует символы строки *S* в целое число. Если строка не содержит правильного представления целого числа, возвращается значение *D*
Function StrToIntRange (S : String; Min, Max : LongInt) : LongInt;	Преобразует символы строки *S* в целое число и возбуждает исключение ERangeError, если число выходит из заданного диапазона Min.. Max
Function StrToTime (S : String) : TDateTime;	Преобразует символы строки *S* во время. При этом строка должна содержать два или три числа, разделенных правильным для Windows разделителем времени. Числа задают часы, минуты и, возможно, секунды. За последним числом через пробел могут следовать символы «am» или «pm», указывающие на 12-часовой формат времени

Продолжение табл. П3.2

Интерфейс	Назначение
Procedure Val (S : String; Var X; C : Integer);	Преобразует строку символов *S* во внутреннее представление целой или вещественной переменной *X*, которое определяется типом этой переменной. Параметр *C* содержит нуль, если преобразование прошло успешно, и тогда в *X* помещается результат преобразования. В противном случае параметр содержит номер позиции в строке *S*, где обнаружен ошибочный символ, и в этом случае содержимое *X* не изменяется. В строке *S* могут быть ведущие и (или) ведомые пробелы
Подпрограммы приведения к строковому типу	
Function DateTimeToStr (Value : TDateTime) : String;	Преобразует дату и время из параметра Value в строку символов
Procedure DateTimeToString (Var S : String; Format : String; Value : TDataTime);	Преобразует дату и время из параметра Value в строку *S* в соответствии со спецификаторами параметра Format
Function DateToStr (Value : TDateTime) : String;	Преобразует дату из параметра Value в строку символов
Function FloatToStr (Value : Extended) : String;	Преобразует вещественное значение Value в строку символов
Function FloatToStrF (Value : Extended; Format : TFloatFormat; Precision, Digits : Integer) : String;	Преобразует вещественное значение Value в строку символов с учетом параметра Format и параметров Precision и Digits
Function Format (const Format : String; const Args : array of const) : String;	Преобразует произвольное число аргументов открытого массива Args в строку в соответствии со спецификаторами параметра Format
Function FormatDateTime (Format : String; Value : TDateTime) : String;	Преобразует дату и время из параметра Value в строку символов в соответствии со спецификаторами параметра Format

Окончание табл. П3.2

Интерфейс	Назначение
Function FormatFloat (Format : String; Value : Extended) : String;	Преобразует вещественное число Value в строку символов в соответствии со спецификаторами параметра Format
Function IntToHex (Value : Integer; Digits : Integer) : String;	Преобразует целое число Value в строку символьного представления шестнадцатеричного формата: Digits — минимальное число символов в строке
Function IntToStr (Value : Integer) : String;	Преобразует целое число Value в строку символов
Procedure Str (X [:Width [:Decimals]]; Var S : String);	Преобразует число *X* любого вещественного или целого типа в строку символов *S*; параметры Width и Decimals, если они присутствуют, задают формат преобразования: Width определяет общую ширину поля, выделенного под соответствующее символьное представление числа *X*, а Decimals — число символов в дробной части (этот параметр имеет смысл только в случае если *X* — вещественное число)
Function TimeToStr (Value : TDateTime) : String;	Преобразует время из параметра Value в строку символов

Таблица П3.3. Подпрограммы для работы с файлами

Интерфейс	Назначение
Procedure AssignFile (Var F; FileName : String);	Связывает файловую переменную *F* с именем файла FileName
Procedure CloseFile (Var F);	Закрывает файл, при этом связь файловой переменной *F* с именем файла, установленная ранее процедурой AssignFile, сохраняется
Function EOF (Var F) : Boolean;	Тестирует конец файла и возвращает значение True, если файловый указатель стоит в конце

Продолжение табл. П3.2

Интерфейс	Назначение
	файла. При записи это означает, что очередной компонент будет добавлен в конец файла, а при считывании — что файл исчерпан
Procedure Erase (Var F);	Уничтожает файл *F*, причем перед выполнением процедуры его необходимо закрыть. В ряде случаев вместо процедуры Erase удобнее использовать функцию DeleteFile, которая не требует предварительного связывания имени файла с файловой переменной
Function FileExists (const FileName : String) : Boolean;	Возвращает значение True, если файл с именем FileName (и, возможно, с маршрутом доступа), существует
Function FindFirst (const Path : String; Attr : Integer; Var F : TSearchRec) : Integer;	Возвращает атрибуты первого из файлов, зарегистрированных в указанном каталоге: Path — маршрут поиска и маска выбора файлов; Attr — атрибуты выбираемых файлов; *F* — переменная типа TSesrchRec, в которой будет возвращено имя первого выбранного файла. При успешном поиске возвращает значение 0
Procedure FindClose (Var F : TSearchRec);	Освобождает память, выделенную для поиска файлов функциями FindFirst и FindNext
Function FindNext (Var F : TSearchRec) : Integer;	Возвращает в переменой *F* имя следующего файла в каталоге, причем переменная *F* должна предварительно инициализироваться обращением к функции FindFirst. При успешном поиске возвращает значение 0
Procedure Flush (Var F);	Очищает внутренний буфер файла и таким образом гарантирует сохранность всех последних изменений файла на диске

Окончание табл. П3.2

Интерфейс	Назначение
Procedure GetDir (D : Byte; Var S : String);	Возвращает имя текущего каталога (каталога по умолчанию): *D* — номер устройства (0 — устройство по умолчанию, 1 — диск A, 2 — диск *B* и т. д.); *S* — переменная типа String, в которой возвращается путь к текущему каталогу на указанном диске
Procedure MkDir (Dir : String);	Создает новый каталог на указанном диске: Dir — маршрут поиска каталога. Последним именем в маршруте, т. е. именем вновь создаваемого каталога, не может быть имя уже существующего каталога
Procedure Rename (Var F; S : String);	Переименовывает файл *F* (*S* — строковое выражение, содержащее новое имя файла), причем перед выполнением процедуры необходимо закрыть файл
Procedure Reset (Var F : File; RecSize : Word]);	Открывает существующий файл (RecSize имеет смысл только для нетипизированных файлов) и определяет размер блока данных
Procedure Rewrite (Var F : File [; Recsize: Word]);	Создает новый файл (RecSize имеет смысл только для нетипизированных файлов) и определяет размер блока данных
Procedure RmDir (Dir : String);	Удаляет каталог Dir, причем удаляемый каталог должен быть пустым, т. е. не должен содержать файлов или имен каталогов нижнего уровня

Список литературы

1. *Абрамов В. Г.* Введение в язык Паскаль / В. Г. Абрамов, Н. П. Трифонов, Г. Н. Трифонова. — М. : Наука, 1988.

2. *Ван Тассел Д.* Стиль, разработка, эффективность, отладка и испытание программ / Д. Ван Тассел. — М. : Мир, 1981.

3. *Вирт Н.* Алгоритмы и структуры данных / Н. Вирт. — М. : Мир, 1989.

4. *Гладков В. П.* Задачи по информатике на вступительном экзамене в ВУЗ и их решения / В. П. Гладков. — Пермь : Изд-во Перм. техн. ун-та, 1994.

5. *Грогоно П.* Программирование на языке Паскаль / П. Грогоно. — М. : Мир, 1982.

6. *Дагене В. А.* 100 задач по программированию / В. А. Дагене, Г. К. Григас, К. Ф. Аугутис. — М. : Просвещение, 1993.

7. Delphi: книга рецептов. Практические примеры, трюки и секреты / пер. с чеш. ; под ред. М. В. Финикова, О. И. Березкиной. — (Серия «Просто о сложном»). — СПб. : Наука и техника, 2006.

8. *Епашников А. М.* Программирование в среде Турбо Паскаль 7.0 / А. М. Епашников, В. А. Епашников. — М. : МИФИ, 1994.

9. Задачи по программированию / [С. А. Абрамов, Г. Г. Гнездилова, Е. Н. Капустина и др.]. — М. : Наука, 1988.

10. *Зубов В. С.* Программирование на языке Turbo Pascal (версии 6.0 и 7.0) / В. С. Зубов. — М. : Информационно-издательский дом «Филинъ», 1997.

11. *Иванова Г. С.* Объектно-ориентированное программирование : учебник для вузов / Г. С. Иванова, Т. Н. Ничушкина, Е. К. Пугачев ; под ред. Г. С. Ивановой. — М. : Изд-во МГТУ им. Н. Э. Баумана, 2001.

12. Информатика. Задачник-практикум : в 2 т. / под ред. И. Семакина, Е. Хеннера. — М. : Лаборатория Базовых Знаний, 1999.

13. *Истомин Е. П.* Программирование на алгоритмических языках высокого уровня / Е. П. Истомин, С. Ю. Неклюдов. — СПб. : Изд-во Михайлова В. А., 2003.

14. *Йенсен К.* Паскаль — руководство для пользователей и описание языка / К. Йенсен, Н. Вирт. — М. : Мир, 1982.

15. *Касаткин В. Н.* Информация. Алгоритмы. ЭВМ / В. Н. Касаткин. — М. : Просвещение, 1991.

16. *Культин Н. Б.* Delphi в задачах и примерах / Н. Б. Культин. — СПб. : БХВ-Петербург, 2005.

17. *Ляхович В. Ф.* Руководство к решению задач по основам информатики и вычислительной техники / В. Ф. Ляхович. — М. : Высш. шк., 1994.

18. *Марченко А. И.* Программирование в среде Turbo Pascal 7.0 / А. И. Марченко, Л. А. Марченко ; под ред. В. П. Тарасенко. — Киев : ВЕК+, М. : Бином Универсал, 1998.

19. *Миков А. И.* Информатика. Введение в компьютерные науки / А. И. Миков. — Пермь : Изд-во ПГУ, 1998.

20. *Пильщиков В. Н.* Сборник упражнений по языку Паскаль / В. Н. Пильщиков. — М. : Наука, 1989.

21. *Попов Б. В.* TURBO PASCAL для школьников. Версия 7.0 / Б. В. Попов. — М. : Финансы и статистика, 1996.

22. *Фаронов В. В.* Delphi. Программирование на языке высокого уровня : учебник для вузов / В. В. Фаронов. — СПб. : Питер, 2003.

23. *Хонсбергер Р.* Математические изюминки / Р. Хонсбергер. — М. : Наука, 1992.

24. *Шень А.* Программирование: теоремы и задачи / А. Шень. — М. : МЦНМО, 1995.

25. *Шупруга В. В.* Delphi 2006 на примерах / В. В. Шупруга. — СПб. : БХВ-Петербург, 2006.

Оглавление

Учебное издание

Семакин Игорь Геннадьевич,
Шестаков Александр Петрович

Основы алгоритмизации и программирования

Учебник

Редактор *Л. В. Толочкова*
Технический редактор *Е. Ф. Коржуева*
Компьютерная верстка: *Р. Ю. Волкова*
Корректор *И. А. Ермакова*

Изд. № 103116273. Подписано в печать 25.02.2016. Формат 60×90/16.
Гарнитура «Балтика». Бумага офсетная № 1. Печать офсетная. Усл. печ. л. 19,0.
Тираж 1 000 экз. Заказ № М-1031.

ООО «Издательский центр «Академия». www.academia-moscow.ru
129085, Москва, пр-т Мира, 101В, стр. 1.
Тел./факс: (495) 648-0507, 616-00-29.
Санитарно-эпидемиологическое заключение № РОСС RU. АЕ51. П 16679 от 25.05.2015.

Отпечатано в полном соответствии с качеством
предоставленного электронного оригинал-макета
в типографии филиала АО «ТАТМЕДИА» «ПИК «Идел-Пресс».
420066,г. Казань,ул. Декабристов,2.
E-mail: idelpress@mail.ru

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

ISBN 978-5-4468-3155-5

Издательский центр «Академия»
www.academia-moscow.ru