Голицына О.Л. Попов И.И

программирование на языках высокого уровня

О. Л. Голицына,
И. И. Попов

ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ ВЫСОКОГО УРОВНЯ

Рекомендовано Методическим советом Учебно-методического центра по профессиональному образованию Департамента образования города Москвы в качестве учебного пособия для студентов образовательных учреждений среднего профессионального образования

МОСКВА
2010

УДК 004.2(075.32)
ББК 32.973-02я723
Г60

Рецензенты:
директор института инновационного менеджмента МИФИ,
кандидат технических наук, доцент *И. В. Прохоров*;
доктор технических наук, профессор, зав. кафедрой информационных систем в экономике и менеджменте РЭА им. Г. В. Плеханова *В. П. Романов*;
руководитель отдела информационных и образовательных технологий Учебно-методического центра по профессиональному образованию Департамента образования города Москвы, кандидат педагогических наук *Н. Л. Демкина*

Голицына О. Л., Попов И. И.

Г60 Программирование на языках высокого уровня : учебное пособие / О. Л. Голицына, И. И. Попов. — М. : ФОРУМ, 2010. — 496 с. : ил. — (Профессиональное образование).

ISBN 978-5-91134-209-8

Рассмотрены основные принципы программирования на языках высокого уровня: основные управляющие структуры программирования; стандартные типы данных; структуры данных; процедуры и функции; модульные программы; рекурсивные определения и алгоритмы; вопросы спецификации программ; корректность и надежность программ. Описаны методологии программирования — императивная, объектно-ориентированная, функциональная и логическая, представлены примеры алгоритмических языков. Приведены характеристики инструментальных средств разработки программ. Рассмотрен язык программирования Object Pascal и интегрированная среда разработки программ Delphi, включая архитектуру приложений, работающих с внешними источниками данных.

Для студентов, специализирующихся в области информационных технологий и разработки программных средств.

УДК 004.2(075.32)
ББК 32.973-02я723

ISBN 978-5-91134-209-8

Введение

Информационные технологии в настоящее время являются одной из наиболее быстро развивающихся областей современной жизни. Развитие информационных технологий обусловливает и развитие программного обеспечения (ПО), как одной из важнейших своих составляющих.

В создании ПО значительную роль играет уровень возможностей средств общения человека с ЭВМ. Во многом этот уровень определяется языками программирования (ЯП).

В развитии вычислительной техники и информационных технологий аппаратное и программное обеспечение оказывали друг на друга постоянное взаимное влияние. С ростом вычислительных мощностей повышался и уровень абстракции при записи программы, задающей определенные последовательности действий.

Первые ЯП представляли собой набор машинных команд в двоичном (бинарном) или восьмеричном формате, который определялся архитектурой конкретной ЭВМ. Каждый тип ЭВМ имел свой ЯП, программы на котором были пригодны только для данного типа ЭВМ. От программиста при этом требовалось хорошее знание не только машинного языка, но и архитектуры ЭВМ.

Следующий этап развития ЯП характеризуется созданием языков ассемблерного типа (*ассемблеров*, *макроассемблеров*), позволяющих вместо двоичных и других форматов машинных команд использовать их мнемонические символьные обозначения (имена). Являясь существенным шагом вперед, ассемблерные языки все еще оставались машинно-зависимыми, а программист все также должен быть хорошо знаком с организацией и функционированием аппаратной среды конкретного типа ЭВМ. При этом ассемблерные программы также затруднительны для чтения, трудоемки при отладке и требуют больших усилий для переноса на другие типы ЭВМ.

Языки высокого уровня (ЯВУ) открывают третье поколение ЯП. Основная черта высокоуровневых языков — введение смысловых конструкций, кратко описывающих целые структуры данных и операции над ними. Первый ЯВУ — Fortran — был разработан под руководством Дж. Бэкуса в фирме IBM в 1956 г.

ЯВУ характеризовались следующими чертами (*Sammet J. E.* The State of Programming Language. In: Information technology / J. Monet(ed). North Holland, 1978):

- от пользователя не требуется знания машинного языка;
- язык не связан с определенным типом ЭВМ, поэтому обеспечивает перевод программ с одной ЭВМ на другую;
- одна инструкция ЯВУ переводится в несколько команд машинного кода;
- выражения языка соответствуют области его применения и представляют собой жесткую табличную форму.

Существенное сокращение зависимости программ от аппаратного обеспечения было достигнуто за счет появления специализированных *программ-трансляторов*, преобразующих инструкции ЯВУ в коды конкретной вычислительной машины. Некоторая потеря в скорости вычислений при этом компенсировалась выигрышем в скорости разработки ПО и унификацией программного кода.

Программа на первых языках высокого уровня представляла собой набор инструкций (директив), обращенных к компьютеру. Такой подход к программированию получил название *императивного.*

Важной особенностью ЯВУ стала возможность повторного использования ранее написанных программных блоков (подпрограмм), выполняющих те или иные действия, путем их идентификации и последующего обращения к ним по имени. Такие блоки получили название *процедур* или *функций.*

Дальнейшее развитие ЯВУ привело к появлению нового подхода к программированию — *декларативного* (непроцедурного), определяемого тем, что программы формируют не последовательности шагов выполнения алгоритмов, а содержат описание действий.

В декларативном подходе выделяются две ветви — функциональное и логическое программирование.

Отличительной особенностью языков *функционального* программирования является то, что любая программа, написанная на таком языке, может интерпретироваться как функция с од-

ним или несколькими аргументами. Такой подход дает возможность моделировать программу математическими средствами. Сложные программы строятся посредством агрегирования функций. При этом программа представляет собой функцию, некоторые аргументы которой можно также рассматривать как функции. Таким образом, повторное использование кода сводится к вызову ранее описанной функции, структура которой, в отличие от процедуры императивного языка, математически прозрачна.

Согласно *логическому* подходу программа представляет собой совокупность *правил* или *логических высказываний*. Языки логического программирования базируются на классической логике и применимы для систем *логического вывода* (например, для экспертных систем).

Следующим определяющим шагом совершенствования ЯВУ стало появление *объектно-ориентированного* подхода к программированию (ООП) и соответствующего класса языков. Понятие программного объекта впервые было использовано в языке Simula-67. При ООП программа представляет собой описание совокупностей (классов) объектов, их свойств (атрибутов), отношений между ними, а также операций над объектами (методов). Механизм наследования атрибутов и методов позволяет строить производные понятия на основе базовых и, таким образом, создавать модель определенной предметной области с заданными свойствами. Кроме того, использование разработанных ранее библиотек объектов и методов дает возможность значительно сэкономить трудозатраты при создании программного обеспечения.

В настоящее время, пожалуй, эти четыре подхода (или методологии) можно рассматривать как базовые при описании семантики ЯВУ. На самом деле, в развитии ЯВУ (особенно начиная с четвертого поколения — 4GL) можно наблюдать несколько направлений.

С появлением таких абстракций, как структуры данных, возникают языки программирования, ориентированные на обработку определенных структур. Это и императивный стековый язык Forth, и декларативный Lisp (для работы со списками и анализа текстов).

Расширение возможностей печатающих устройств привело к разработке специализированных языков обработки данных (или управления устройством). Примером такого языка может служить PostScript, созданный на базе языка Forth.

Мощные интегрированные среды разработки с удобным пользовательским интерфейсом, появившиеся с развитием ООП, начинают использоваться для поддержки проблемно-ориентированных языков, оперирующих конкретными понятиями узкой предметной области. Примером здесь могут служить встроенные языки СУБД (FoxPro).

Еще одним направлением развития языков четвертого поколения можно считать языки запросов, позволяющие пользователю получать информацию из баз данных. Языки запросов имеют особый синтаксис, соблюдение которого необходимо, как и в традиционных ЯВУ третьего поколения. Среди языков запросов фактическим стандартом стал язык SQL (Structured Query Language).

Необходимость создания программного обеспечения для вычислительных средств параллельной архитектуры (многомашинных, мультипроцессорных сред и др.) привела к разработке языков параллельного программирования (Occam, SISAL, Oz и др.). Программы, написанные на таких языках, представляют собой совокупность описаний процессов, которые могут выполняться как одновременно, так и в псевдопараллельном режиме, когда устройство, обрабатывающее процессы, функционирует в режиме разделения времени (выделяя время на обработку данных, поступающих от процессов, по мере необходимости).

С развитием Internet-технологий и событийно управляемой концепции ООП в 90-х годах XX в. появился целый класс ЯВУ, которые получили название языков сценариев или *скриптов*. Эти языки первоначально ориентировались на использование в качестве внутренних управляющих языков в сложных системах. Программа на таком языке представляет собой совокупность возможных сценариев обработки данных, выбор которых инициируется наступлением того или иного события. Характерными особенностями данных языков являются интерпретируемость (компиляция либо невозможна, либо нежелательна), простота синтаксиса и легкая расширяемость. Наиболее часто используемые из таких языков — JavaScript (язык для описания поведения веб-страниц интерпретируется браузером во время ее отображения); VBScript (является усеченной версией языка Visual Basic, исполняется браузером при отображении веб-страницы); Perl (является интерпретируемым языком и реализован практически на всех существующих платформах, применяется при обработке текстов, а также для динамической генерации веб-страниц на веб-серверах); Python (по структуре и области применения близок к Perl).

Перечень направлений может быть не полон, поскольку языки программирования (вместе с информационными технологиями) постоянно развиваются и совершенствуются с появлением новых инструментальных средств и теоретических обоснований. При многообразии возможностей выбор языка программирования и средств разработки для решения той или иной задачи даже для специалиста становится самостоятельной проблемой. Тем не менее представляется возможным в рамках четырех базовых методологий сформулировать основные понятия и принципы программирования на ЯВУ, которые могут определить степень пригодности языка для решения поставленной задачи. В соответствии с определяющими признаками этих методологий и организована структура данного учебного пособия.

В первой главе рассмотрены основные управляющие элементы программирования; вопросы, связанные с понятием типов и структур данных, использованием процедур и функций, модульным программированием, определением рекурсивных алгоритмов, задачами стандартизации ЯП и спецификации программ.

Вторая глава посвящена методологиям программирования. В рамках этой главы описываются вычислительные модели императивного, объектно-ориентированного, функционального и логического программирования. В качестве примера реализации каждой методологии приведено краткое описание поддерживающего ее языка программирования.

В третьей главе рассмотрены вопросы обеспечения жизненного цикла, качества и надежности программных средств; приведены краткие характеристики инструментальных сред разработки программ, CASE-средств, а также современных методов и языков моделирования программных систем; представлены некоторые принципы разработки графических интерфейсов.

В четвертой главе дается описание ЯВУ Object Pascal и интегрированной среды разработки приложений Delphi. Эта среда на сегодняшний день является одной из самых популярных и используемых в вузах при изучении методов высокоуровневого программирования. В рамках главы представлены синтаксис и семантика языка Object Pascal, интерфейс среды Delphi, характеристика проекта Delphi, компиляция и выполнение проекта, порядок разработки приложения. Приведена характеристика архитектуры приложений, работающих с внешними источниками данных.

Глава 1
ОСНОВНЫЕ ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКАХ ВЫСОКОГО УРОВНЯ

Компьютер (в прикладном аспекте) используется для выполнения разнообразных функций: работы с текстами, создания графических изображений, получения справок из баз данных, проведения табличных расчетов, решения математических задач и т. п. В настоящее время в распоряжении пользователя ЭВМ имеется программное обеспечение (ПО), традиционно подразделяющееся на следующие классы:

- системное ПО, ядром которого является операционная система;
- прикладное ПО, включающее в основном функционально-ориентированные программы;
- инструментальное ПО, в состав которого входят средства для создания программ на языках программирования.

Исходя из условия поставленной задачи, пользователь решает для себя вопрос о том, каким программным средством он воспользуется. Если в составе доступного прикладного ПО есть программа, подходящая для решения данной задачи, то пользователь выбирает ее в качестве инструмента (СУБД, табличный процессор, математический пакет и др.). В случае же, когда готовым программным продуктом воспользоваться нельзя, прибегают к программированию — теоретической и практической деятельности, связанной с созданием программ на универсальных языках программирования.

1.1. Этапы решения задач на ЭВМ

Работа по решению прикладной задачи на компьютере проходит через следующие этапы:

- постановка задачи;
- математическая формализация;

- построение алгоритма;
- составление программы на языке программирования;
- отладка и тестирование программы;
- анализ полученных результатов.

Технологическая цепочка решения задачи на ЭВМ предусматривает возможность возвратов на предыдущие этапы после анализа полученных результатов (рис. 1.1). Часто в эту цепочку включают еще один пункт: составление сценария интерфейса (т. е. взаимодействия между пользователем и компьютером во время исполнения программы).

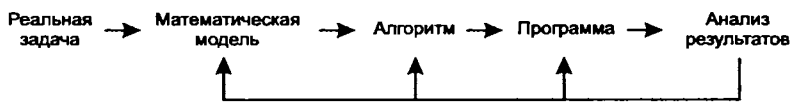


Рис. 1.1. Технологическая цепочка решения задачи на ЭВМ

Рассмотрим каждый из этапов.

Постановка задачи

Этап постановки задачи включает:
- формулировку условия задачи;
- определение конечных целей решения задачи;
- описание исходных данных (их типов, диапазонов возможных значений, структуры и т. п.);
- определение формы выдачи результатов.

На этом этапе должно быть четко определено, что известно и что требуется получить в результате. Рассмотрим, например, задачу нахождения решения системы двух линейных алгебраических уравнений (если оно существует и единственно):

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1; \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{cases} \quad \text{или} \quad \mathbf{A} \times \mathbf{x} = \mathbf{b}.$$

Постановка такой задачи выглядит следующим образом.

Дано: $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ — матрица коэффициентов; $\mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$ — вектор свободных членов.

Необходимо найти вектор-столбец неизвестных (переменных) $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, где x_1, x_2 — решение системы.

Исходные данные (элементы матрицы A и вектора b) — произвольные действительные числа.

Выдача результата должна содержать найденные значения x_1 и x_2 или сообщение о том, что единственного решения не существует.

Математическая формализация

Построение математической модели означает формализацию способа получения результата из исходных данных, опирается на анализ существующих аналогов и анализ технических и программных средств и включает следующую последовательность шагов:

- разработку математической модели — формального выражения связи между исходными данными и результатом;
- разработку структур данных, поддерживающих преобразование исходных данных в результат.

Компьютер, как формальное вычислительное устройство, решает задачу, выполняя команды, выраженные на языке программирования. Выполнение операций происходит в компьютере по законам двоичной системы счисления и булевой алгебры. Это становится возможным, если все необходимые для решения задачи действия формализованы, т. е. представлены как последовательность операций (математических, логических, сравнения) над переменными.

Разработка математической модели для поставленной задачи решения системы из двух линейных алгебраических уравнений состоит в описании в виде математических выражений процесса поиска решения.

1. Из курса линейной алгебры известно, что система линейных уравнений имеет единственное решение тогда и только тогда, когда определитель матрицы системы отличен от нуля. Это условие записывается следующим образом:

$$\Delta = a_{11} \cdot a_{21} - a_{21} \cdot a_{12} \neq 0.$$

2. Если условие существования и единственности решения выполняется, то для поиска решения системы можно использовать следующие формулы (формулы Крамера):

$$x_1 = \frac{\Delta_1}{\Delta} = \frac{b_1 \cdot a_{22} - b_2 \cdot a_{12}}{a_{11} \cdot a_{22} - a_{21} \cdot a_{12}}; \quad x_2 = \frac{\Delta_2}{\Delta} = \frac{b_2 \cdot a_{11} - b_1 \cdot a_{21}}{a_{11} \cdot a_{22} - a_{21} \cdot a_{12}}.$$

Разработка структур данных сводится к выбору структуры для размещения вводимых пользователем коэффициентов системы и свободных членов.

Построение алгоритма

Этап построения алгоритма предполагает формирование строгой и четкой системы правил, определяющей последовательность действий, которая за конечное число шагов должна привести к результату. В этот этап входят:

- выбор формы записи алгоритма (естественный язык, блок-схема, псевдокод);
- проектирование алгоритма.

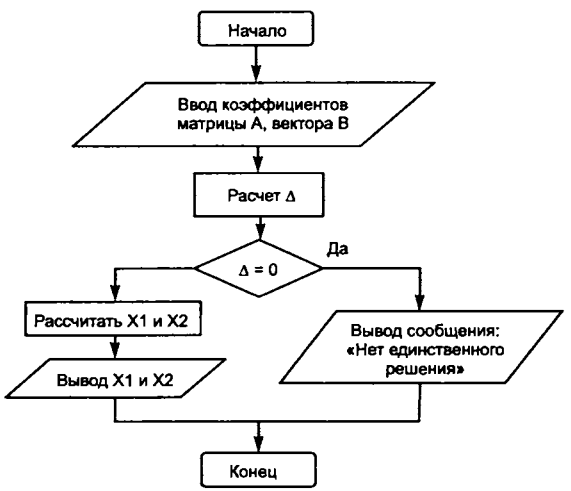


Рис. 1.2. Блок-схема плгоритма решения системы линейных уравнений

Выбрав блок-схему как форму записи алгоритма, получаем алгоритм решения поставленной задачи, представленный на рис. 1.2.

Составление программы на языке программирования

Этап составления программы включает следующие шаги:

- выбор языка программирования;
- уточнение способов организации данных;
- запись алгоритма на выбранном языке программирования.

Предположим, что для записи алгоритма выбран учебный язык Basic. Тогда программа поиска решения системы может быть, например, такой:

```
DIM A(2,2), B(2)      'описание исходной матрицы A и вектора B
For I% = 1 to 2
    For J% = 1 to 2   'Ввод элементов матрицы
     Input("Введите коэффициент A["+Str(I%)+","+ Str(J%)+"]:",
                                        A[I%,J%])
    Next J%
Next I%
For I% = 1 to 2       'Ввод свободных членов
    Input("Введите свободный член B["+Str(I%)+"]:", B[I%]);
Next I%
DELTA = A[1,1]*A[2,2]-A[1,2]*A[2,1] 'Расчет определителя
If DELTA = 0 Then
    Print ("Нет единственного решения")    'Вывод сообщения
Else
    X1 = (B[1]*A[2,2]-b[2]*A[1,2])/DELTA
    X2 = (B[2]*A[1,1]-b[1]*A[2,1])/DELTA
    Print ("X1 = ", X1)    'Вывод значения X1
    Print ("X2 = ", X2)    'Вывод значения X2
End If
End
```

Отладка и тестирование программы

Отладка программы предполагает следующие действия:

- синтаксическая отладка;
- отладка семантики и логической структуры;
- тестовые прогоны;
- анализ результатов тестирования.

Под отладкой программы понимается процесс испытания работы программы и исправления обнаруженных при этом ошибок. Обнаружить ошибки, связанные с нарушением правил записи программы на ЯПВУ (синтаксические и семантические ошибки), помогает используемая система программирования. Пользователь получает сообщение об ошибке, исправляет ее и снова повторяет попытку выполнить программу.

Проверка на компьютере правильности алгоритма проводится с помощью тестов. Тестом, например, можно назвать конкретный вариант значений исходных данных, для которого известен ожидаемый результат. Корректное выполнение программой любого теста — необходимое, но не достаточное условие ее правильности. Например, для программы решения системы ли-

нейных уравнений необходимо построить тесты, позволяющие проверить работоспособность как для варианта, когда определитель матрицы **A** равен 0, так и для варианта, когда решение системы существует и единственно.

Анализ получаемых результатов

Последний этап — применение разработанной программы для получения искомых результатов. На этом этапе могут быть сделаны выводы о некорректности постановки задачи или разработанной математической модели. Например, при построении математической модели для задачи решения системы линейных уравнений мы не учитывали погрешность, которая может возникать при выполнении операций над очень большими числами. В этом случае происходит откат на этап постановки задачи или на этап математической формализации, что приводит иногда к повторной разработке программы.

Программы, имеющие большое практическое или научное значение, используются длительное время. Иногда в процессе эксплуатации программы исправляются, дорабатываются, поэтому важным этапом жизни программ является их *сопровождение,* включающее при необходимости доработку программы для решения новых задач, а также составление документации не только по использованию программы, но и по математической модели, алгоритму, набору тестов и т. п.

1.2. Представление основных управляющих структур программирования

Для описания синтаксических конструкций языков программирования в настоящем пособии используются две нотации:

- Бэкуса (впервые предложена при описании языка ALGOL);
- IBM (разработана фирмой для описания языков COBOL и JCL).

Нотация Бэкуса содержит конструкции следующего вида:

```
<Оператор присваивания> ::= <Переменная> := <Выражение>
<Слово> ::= <Буква>|<Слово><Буква>
```

Левая часть определения конструкции языка содержит наименование определяемого элемента, взятого в угловые скобки.

Правая часть включает совокупность элементов, соединенных знаком |, который трактуется как «или» и объединяет альтернативы — различные варианты значения определяемого элемента.

Части соединяются оператором ::=, который означает *есть по определению.*

Определение может носить рекурсивный характер, т. е. правая часть конструкции может содержать объект из левой части (как для объекта <Слово>). В этом случае вместо объектов в правой части определения можно подставлять любые их значения. В соответствии с приведенным определением объекта <Слово> любая последовательность букв (например, латинского алфавита) может быть названа словом: A, AA, AAB и т. п.

Нотация IBM включает следующие конструкции:

< > — *угловые скобки* (или двойные кавычки " ") обозначают элементы программы, определяемые пользователем (<идентификатор>, <список параметров> <условие> и пр. В соответствующих местах реальной программы будет находиться идентификатор переменной и т. д.);

[ ] — *квадратные скобки*, ограничивающие синтаксическую конструкцию, обозначают ее возможное отсутствие. Например, return [<выражение>]; В этой конструкции <выражение> не обязательно;

| — вертикальная черта разделяет список значений обязательных элементов, одно из которых должно быть выбрано;

... — *горизонтальное* многоточие, следующее после некоторой синтаксической конструкции, обозначает последовательность конструкций той же самой формы, что и предшествующая многоточию конструкция. Например, ={<выражение> [,<выражение>]...} обозначает, что одно или более выражений, разделенных запятыми, может появиться между фигурными скобками.

Например:

```
[Private|Public|Friend] [Static] Sub <идентификатор>
     [(<список параметров>)]
[<оператор>]
...
[Exit Sub]
[<оператор>]
...
End Sub
```

Рекурсивные определения в IBM-нотации не используются.

1.2.1. Базовые структуры алгоритмов

Графическая запись алгоритма (блок-схема) представляет собой композицию алгоритмических структур, называемых базовыми. Базовые алгоритмические структуры — это определенные наборы блоков и стандартных способов их соединения для выполнения типичных последовательностей действий. Выделяют следующие типичные последовательности действий: линейную, разветвляющуюся и циклическую.

Линейными называются алгоритмы, в которых действия осуществляются последовательно друг за другом. Стандартная блок-схема линейного алгоритма (вычисление произведения двух чисел A и B) приводится на рис. 1.3.

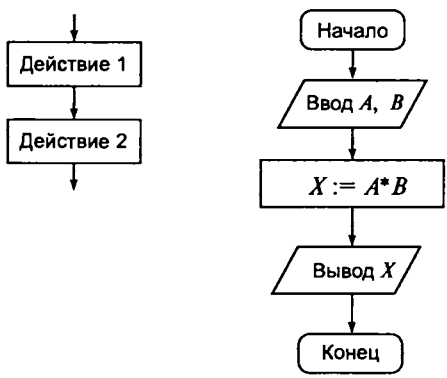


Рис. 1.3. Линейный алгоритм

Разветвляющимся называется алгоритм, в котором действие выполняется по одной из возможных ветвей решения задачи, в зависимости от выполнения условий. В отличие от линейных алгоритмов, в которых действия выполняются последовательно, в разветвляющиеся алгоритмы входит условие, в зависимости от выполнения или невыполнения которого производится та или иная последовательность действий.

В качестве условия в разветвляющемся алгоритме может быть использовано любое понятное исполнителю утверждение, которое может соблюдаться (быть истинно) или не соблюдаться (быть ложно). Такое утверждение может быть выражено как словами, так и формулой. Таким образом, алгоритм ветвления состоит из условия и двух последовательностей команд.

Примером может являться разветвляющийся алгоритм, изображенный на рис. 1.4. Аргументами этого алгоритма являются числа *A* и *B*, а результатом — число *X*. Если условие *A* >= *B* истинно, то выполняется операция умножения чисел (*X* = *A***B*), в противном случае — операция сложения (*X* = *A* + *B*). В результате печатается то значение *X*, которое получается после выполнения одного из действий.

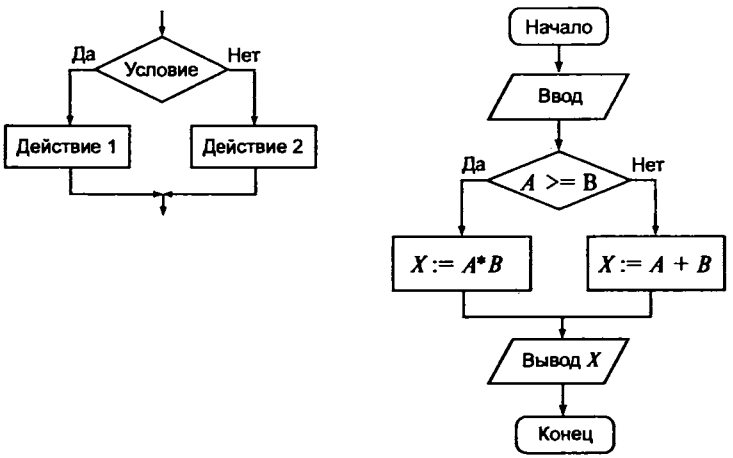


Рис. 1.4. Разветвляющийся алгоритм

Циклическим называется алгоритм, в котором некоторая часть операций (тело цикла — последовательность команд) выполняется многократно. Однако слово «многократно» не значит «до бесконечности». Организация циклов, никогда не приводящая к остановке в выполнении алгоритма, является нарушением требования получения результата за конечное число шагов.

Перед операцией цикла осуществляются операции присвоения начальных значений тем объектам, которые используются в теле цикла. В цикл входят в качестве базовых следующие структуры: блок проверки условия и блок, называемый *телом цикла*. Если тело цикла расположено после проверки условий (рис. 1.5, *а* — цикл с предусловием), то может случиться, что при определенных условиях тело цикла не выполнится ни разу. Такой вариант организации цикла, управляемый предусловием, называется *цикл типа пока* (здесь условие — это условие на продолжение цикла).

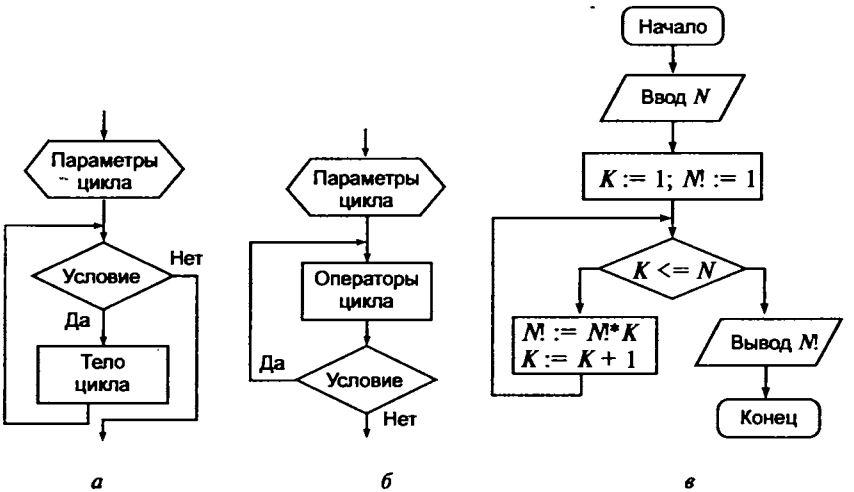


Рис. 1.5. Циклические алгоритмы

Возможен другой случай, когда тело цикла выполняется по крайней мере один раз и будет повторяться до тех пор, пока не станет истинным условие. Такая организация цикла, когда его тело расположено перед проверкой условия, носит название цикла с постусловием, или *цикла типа до* (рис. 1.5, *б*). Истинность условия в этом случае — условие окончания цикла. Отметим, что возможна ситуация с постусловием и при организации цикла *пока*. Итак, цикл-*до* завершается, когда условие становится истинным, а цикл *пока* — когда условие становится ложным. Современные языки программирования имеют достаточный набор операторов, реализующих как цикл *пока*, так и цикл *до*.

Рассмотрим циклический алгоритм *типа пока* (рис. 1.5, *в*) на примере алгоритма вычисления факториала. N — число, факториал которого вычисляется. Начальное значение $N!$ принимается равным 1. K будет меняться от 1 до N и вначале также равно 1. Цикл будет выполняться, пока справедливо условие $N \geq K$. Тело цикла состоит из двух операций $N! = N!*K$ и $K = K + 1$ (рис. 1.5, *в*).

Циклические алгоритмы, в которых тело цикла выполняется заданное число раз, реализуются с помощью цикла со счетчиком. Цикл со счетчиком реализуется с помощью рекурсивного увеличения значения счетчика в теле цикла ($K = K + 1$ в алгоритме вычисления факториала).

1.2.2. Основные программные элементы

Типовые операторы управления вычислительным процессом

Различные типы алгоритмических структур реализуются на языке программирования с помощью операторов. К типовым операторам управления вычислительным процессом относятся следующие:

- операторы присваивания значений (пересылка значений переменных, констант, функций в принимающую переменную; вычисление значений арифметических, логических, символьных переменных в рамках существующих в конкретном языке правил построения выражений);
- организация циклов (выполняемых до исчерпания списка или до достижения управляющей переменной заданного значения, или пока выполняются некоторые условия);
- ветвление программы (выполнение альтернативных групп операторов при заданных условиях);
- блоки операторов (последовательности операторов, выполняемые как целое);
- операторы перехода (условная или безусловная передача управления на определенный оператор, снабженный меткой, или условный/безусловный выход из цикла или блока).

Для кодирования алгоритма на конкретном языке программирования необходимо знать его синтаксис, т. е. синтаксис его основных операторов, типы переменных, правила формирования выражений и т. п.

Каждый оператор в конкретном языке имеет свой формат. В формат операторов, кроме ключевых слов, входят данные и выражения.

Важными составляющими управляющих операторов обычно являются логические условия, сопоставляющие значения данных отношениями типа $\geq$, $\leq$, $>$, $<$, $\neq$ и вырабатывающие решение о продолжении цикла, переходе по ветви, выходе из блока и т. д. Несколько простых условий объединяются в составные посредством логических операций И, ИЛИ, НЕ.

Данные и типы данных

Алгоритм, реализующий решение некоторой конкретной задачи, всегда работает с данными. *Данные* — это любая информа-

ция, представленная в формализованном виде и пригодная для обработки алгоритмом.

Данные делятся на переменные и константы. *Переменные* — это такие данные, значения которых могут изменяться в процессе выполнения алгоритма.

Например, для алгоритма вычисления площади круга необходимо объявить две переменные: переменную R, в которую будет заноситься значение радиуса окружности, и переменную S для вычисления площади круга по формуле:

$$S = \pi R^2.$$

Константы — это данные, значения которых не меняются в процессе выполнения алгоритма. В примере, описанном выше, константой является число π.

Каждая переменная и константа должна иметь свое уникальное *имя*. Имена переменных и констант задаются идентификаторами. *Идентификатор* (по определению) представляет собой последовательность букв и цифр, начинающаяся обычно с буквы.

Аналогично могут быть выделены типичные группы *функций*:

- стандартные алгебраические и арифметические — SIN, COS, SQRT, MIN, MAX и др.;
- стандартные строчные — выделение, удаление подстроки, проверка типа переменной и т. д.;
- нестандартные функции, в том числе: описание операций и форматов ввода-вывода данных; преобразование типов данных; описание операций над данными, специфичными для конкретной системы программирования, ОС или типа ЭВМ.

С данными тесно связано понятие типа данных. Любой константе, переменной, выражению (с точки зрения обработки на ЭВМ) всегда сопоставляется некоторый тип.

Тип данных — это такая характеристика данных, которая, с одной стороны, задает *множество значений* для возможного изменения данных и, с другой стороны, определяет *множество операций*, которые можно к этим данным применять, и правила выполнения этих операций.

Выражения

Выражение — это синтаксическая единица языка, задающая порядок и способ вычисления некоторого значения.

В соответствии с правилами формирования выражение представляет собой последовательность *операндов*, соединяющихся друг с другом знаками *операций*. Некоторые фрагменты выражения могут быть заключены в круглые скобки:

<Выражение> ::= <Операнд> | <Выражение><Бинарная операция> <Выражение> | <Унарная операция><Выражение> | (<Выражение>)

В качестве операнда в конструкции выступают переменные, константы, функции.

Операции подразделяются на несколько групп:

- арифметические операции;
- операции отношения;
- логические операции;
- операции с битами информации.

Каждой группе операций соответствуют определенные типы переменных и констант.

По количеству операндов, участвующих в операциях, их делят на *унарные* (операции с одним операндом) и *бинарные* (операции с двумя операндами). В бинарных операциях используется обычное алгебраическое представление, например A+B. В унарных операциях операция всегда предшествует операнду, например -B.

Арифметические операции. Арифметические операции могут применяться только к операндам целых и вещественных типов (табл. 1.1).

Таблица 1.1. **Арифметические операции**

Знак	Операция	Количествово операндов	Тип операндов	Тип результата	Описание
+	Признак положительного числа	Унарная	Целый	Целый	Не меняет значения операнда
			Вещественный	Вещественный	
-	Признак отрицательного числа	Унарная	Целый	Целый	Меняет знак операнда на противоположный
			Вещественный	Вещественный	
+	Сложение	Бинарная	Целый	Целый	Результат — сумма двух чисел: `25 + 6 = 31`
			Хотя бы один вещественный	Вещественный	

Окончание табл. 1.1

Знак	Операция	Количествово операндов	Тип операндов	Тип результата	Описание
-	Вычитание	Бинарная	Целый	Целый	Результат — разность двух чисел: `25 - 6 = 19`
			Хотя бы один вещественный	Вещественный	
*	Умножение	Бинарная	Цепый	Целый	Результат — произведение двух чисел: `25 * 6 = 150`
			Хотя бы один вещественный	Вещественный	
/	Деление	Бинарная	Целый или вещественный	Вещественный	Результат — частное от деления двух чисел: `25 / 6 = 4,16(6)`
`div`	Деление целых чисел	Бинарная	Целый	Целый	Результат — целая часть от деления целых чисел: `25 div 6 = 4`
`mod`	Остаток от деления целых чисел	Бинарная	Целый	Целый	Результат — остаток от деления целых чисел: `25 mod 6 = 1`

В качестве операндов арифметических операций могут выступать стандартные арифметические (или тригонометрические) функции, т. е. функции с результатом целого или вещественного типа, аргументами которых являются выражения целого или вещественного типа. В табл. 1.2 приведены характеристики некоторых арифметических и тригонометрических функций (список функций приведен для языка Object Pascal).

Таблица 1.2. **Стандартные арифметические (тригонометрические) функции**

Функция	Назначение	Тип аргумента	Тип результата
`Abs(X)`	Модуль (абсолютная величина) аргумента	Целый	Целый
		Вещественный	Вещественный
`Arccos(X)`	Арккосинус аргумента	Вещественный, `Abs(X)<=1`	Вещественный
`Arcsin(X)`	Арксинус аргумента	Вещественный, `Abs(X)<=1`	Вещественный
`ArcTan(X)`	Арктангенс аргумента	Вещественный	Вещественный

Окончание табл. 1.2

Функция	Назначение	Тип аргумента	Тип результата
`Ceil(X)`	Минимальное целое число, большее или равное `X`	Целый или вещественный	Целый. Например: `Ceil(-2.8) = -2` `Ceil(2.8) = 3`
`Cos(X)`	Косинус аргумента	Вещественный	Вещественный
`Cotan(X)`	Котангенс аргумента	Вещественный	Вещественный
`Hypot(X,Y)`	Гипотенуза прямоугольного треугольника с катетами `X`, `Y`	Вещественный	Вещественный
`Log10(X)`	Десятичный логарифм аргумента	Вещественный	Вещественный
`Log2(X)`	Логарифм аргумента по основанию 2	Вещественный	Вещественный
`LogN(N,X)`	Логарифм аргумента `X` по основанию `N`	Вещественные	Вещественный
`Max(A,B)`	Максимальное из чисел `A`, `B`	Оба целые или оба вещественные	Совпадает с типом аргументов
`Min (A,B)`	Минимальное из чисел `A`, `B`	Оба целые или оба вещественные	Совпадает с типом аргументов
`Pi`	Число `p`		Вещественный
`Power(X,Ex)`	Возведение аргумента `X` в степень `Ex`	Вещественные	Вещественный
`Sin(X)`	Синус аргумента	Вещественный	Вещественный
`Sqr(X)`	Квадрат аргумента	Вещественный	Вещественный
`Sqrt(X)`	Квадратный корень аргумента	Вещественный	Вещественный
`Tan(X)`	Тангенс аргумента	Вещественный	Вещественный

Примеры записи арифметических выражений:

```
(Max(X,Y)+sqrt(Z))*2;
Ceil(Tan(X)-1);
Pi* Sqr(R);
(Cos(X)+Y)/Z;
Log2(X)+X/Y;
Power(P,Q)/Power(R,(S+T));
```

Операции отношения. Все операции отношения — бинарные. В результате выполнения таких операций получается значение логического типа: *true* или *false* (табл. 1.3). При этом операнды, участвующие в операции, должны быть сравнимых типов, например: целого и целого; целого и вещественного; логического и логического и т. п. Ошибочно сравнивать операнды символьного и целого типа, или целого и логического.

Таблица 1.3. **Операции отношения**

Операция	Описание	Примеры

=	Равно	`X = Abs(X)` — `true`, если `X` — положительное число, `false`, если `X` — отрицательное
		`A = A` — `true`
		`2 = 5` — `false`

<>	Не равно	`X <> Abs(X)` — `true`, если `X` — отрицательное число, `false`, если `X` — положительное
		`A <> A` — `false`
		`2 <> 5` — `true`

<	Меньше	`A < A` — `false` `2 < 5` — `true`
<=	Меньше или равно	`A <= A` — `true` `Max(X,Y) <= X` — `false`
>	Больше	`A+1 > A` — `true` `Min(X,Y) > X` — `false`
>=	Больше или равно	`A >= A` — `true` `Max(X,Y) >= X` — `true`

Логические операции. Логические операции применяются к операндам логического типа. Результат выполнения логических операций тоже логического типа. Вычисление логических выражений происходит в соответствии с *таблицами истинности* логических операций (табл. 1.4). Таблицы истинности задают соответствие между значениями операндов и результатом выполнения операции.

Конъюнкция (логическое умножение) — бинарная операция соединения двух высказываний в одно с помощью союза И (OR). Эту операцию принято обозначать знаками «∧», «&» или знаком умножения «×». Сложное высказывание `A&B` истинно только в том случае, когда истинны оба входящих в него высказывания.

Таблица 1.4. **Таблицы истинности логических операций с учетом значения Null**

A	*B*	*A* & *B*	*A* ∨ *B*	*A* xor *B*	¬*A*
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false
false	null	false	false	null	true
true	null	null	true	null	false
null	false	false	null	null	null
null	true	null	true	null	null
null	null	null	null	null	null

Дизъюнкция (логическое сложение) — бинарная операция объединения двух высказываний с помощью союза ИЛИ (OR). Эту операцию обозначают знаками «|», «∨» или знаком сложения «+». Сложное высказывание A∨B истинно, если истинно хотя бы одно из входящих в него высказываний.

Исключающее ИЛИ — модифицированная дизъюнкция (XOR), отличающаяся от обычного ИЛИ ложным значением результата при истинности обоих аргументов.

Инверсия (логическое отрицание) — унарная операция присоединения частицы НЕ (NOT) к высказыванию. Она обозначается A (или ¬A) и читается *не* A. Если высказывание A истинно, то ¬A ложно, и наоборот.

Помимо значений `true` и `false` в таблицы истинности операций для некоторых языков программирования добавлено стандартное значение `null`. Как правило, такое значение присваивается компилятором логической переменной по умолчанию. Результат выполнения логической операции со значением `null` в одном или в обоих операндах также может быть равным значению `null`.

Примеры записи логических выражений:

```
(X>=0) and (X<=1)
((X>0) and (X<0.5)) or (X>3)
(N mod 2 = 0) and (N>0)
```

Операции с битами информации. В набор операций некоторых современных ЯП включены операции побитового сравнения содержимого машинных слов, которые могут содержать числовые, строчные и др. данные (табл. 1.5). При этом каждый бит результата вычисляется в соответствии с табл. 1.6 (для бинарных операций). Унарная операция отрицания (`not`) в данном случае реализует очевидную замену 1 на 0 и наоборот.

Таблица 1.5. **Операции с битами информации**

Операция	Описание	Количество операндов
`Not`	Битовое отрицание	Унарная
`And`	И (битовое)	Бинарная
`Or`	ИЛИ (битовое)	Бинарная
`Xor`	Исключающее ИЛИ (битовое)	Бинарная
`Imp`	Импликация	Бинарная
`Eqv`	Эквивалентность	Бинарная
`Shl`	Сдвиг влево	Бинарная
`Shr`	Сдвиг вправо	Бинарная

Таблица 1.6. **Операнды и результаты некоторых операций побитового сравнения**

X	*Y*	*X* and *Y*	*X* or *Y*	*X* Imp *Y*	*X* Eqv *Y*	*X* xor *Y*
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
1	1	1	1	1	1	0

Рассмотрим примеры выполнения операций побитового сравнения. Пусть объявлены две переменных *A* и *B* целого типа. В переменной *A* находится число 12, а в переменной *B* — чис-

ло 9. Запишем значения переменных в двоичном побитовом представлении и применим операции побитового сравнения:

A = 12	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
B = 9	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
Not *B* = not 9 = –10	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0
A and *B* = 12 and 9 = 8	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
A or *B* = 12 or 9 = 13	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1
A eqv *B* = 12 eqv 9 = –6	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
A imp *B* = 12 imp 9 = –5	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
A xor *B* = 12 xor 9 = 5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

В более сложных выражениях порядок, в котором выполняются операции, соответствует приоритету операций (табл. 1.7).

Таблица 1.7. **Приоритет выполнения операций**

Операции	Приоритет	Категория
`not`, `+`, `-`	Перый (высший)	Унарные операции
`*`, `/`, `div`, `mod`, `and`, `shl`, `shr`	Второй	Операции умножения
`+`, `-`, `or`, `xor`	Третий	Операции сложения
`=`, `<>`, `<`, `>`, `<=`, `>=`	Четвертый (низший)	Операции отношения

Для определения старшинства операций применяются три основных правила:

1. Операнд, находящийся между двумя операциями с различными приоритетами, связывается с операцией, имеющей более высокий приоритет.

2. Операнд, находящийся между двумя операциями с равными приоритетами, связывается с операцией, которая находится слева от него.

3. Выражение, заключенное в круглые скобки, перед выполнением вычисляется как отдельный операнд.

Операции с равным приоритетом обычно выполняются слева направо, хотя иногда компилятор при генерации оптимального кода может переупорядочить операнды.

1.2.3. Обработка исключительных ситуаций

Для обеспечения надежности прикладных программ операционные системы предоставляют программисту специальные средства обработки исключительных ситуаций — аппарат обна-

ружения и обработки ошибок и граничных состояний. Наличие подобных средств при написании программного кода позволяет сосредоточиться на решении основной задачи и отделить основную работу от рутинной, хотя и необходимой, обработки ошибок.

Рассмотрим действие такого аппарата на примере *структурной обработки исключений* (Structured Exception Handling — SEH), введенной фирмой Microsoft в операционную систему Win32.

Основная нагрузка при поддержке SEH ложится не на операционную систему, а на компилятор, который должен сгенерировать специальный код на входах и выходах так называемых *блоков исключений* и создать таблицы вспомогательных структур данных для поддержки SEH. Разные фирмы по-разному реализуют SEH в своих компиляторах, но большинство фирм-разработчиков компиляторов придерживается синтаксиса, рекомендованного Microsoft.

SEH предоставляет две основные возможности: *обработку завершения* и *обработку исключений*.

Обрабока завершения

Обработка завершений гарантирует, что некоторый блок кода (обработчик завершения) будет исполнен независимо от того, каким образом произойдет выход из другого блока кода (защищенного блока). Общий синтаксис использования обработки завершения в прикладных программах следующий:

```
try
<защищенный блок>
finally
<обработчик завершения>
```

Ключевые слова `try` и `finally` обозначают два программных блока обработки завершения. Благодаря совместным действиям операционной системы и компилятора гарантируется, что код блока `finally` будет исполнен независимо от того, каким образом произойдет выход из защищенного блока. Порядок исполнения кода при этом следующий.

1. Исполняется код перед блоком `try`.
2. Исполняется код внутри блока `try` (защищенный блок).
3. Исполняется код внутри блока `finally` (обработчик завершения).
4. Исполняется код после блока `finally`.

Рассмотрим пример: пусть необходимо разместить в оперативной памяти и заполнить некоторыми значениями массив целых чисел, провести обработку элементов массива (например, каждый элемент с четным номером возвести в квадрат и посчитать сумму элементов) и освободить оперативную память. Поместив обработку элементов массива в блок `try`, а освобождение памяти — в блок `finally`, можно таким образом обеспечить освобождение памяти при любом исходе, даже если в процессе выполнения алгоритма обработки массива произошла какая-нибудь ошибка. Схема решения такой задачи следующая:

```
<выделение памяти для размещения массива>
<заполнение элементов массива>
try
<выполнение алгоритма обработки>
<вывод результата>
finally
<освобождение занятой памяти>
```

Обработка исключений

Исключение — это не предполагаемое алгоритмом событие. В хорошо написанной программе обычно не должно быть обращений по недопустимому адресу памяти или деления на нуль, но все же такие ошибки иногда случаются. За перехват попыток обращения по недопустимому адресу и деления на нуль отвечает центральный процессор, возбуждающий исключения в ответ на такие ошибки. Исключение, возбуждаемое процессором, называется *аппаратным исключением*. Операционная система и прикладная программа способны также вызывать и *программные исключения*.

При возбуждении аппаратного или программного исключения операционная система дает возможность прикладной программе определить тип исключения и самостоятельно его обработать. Синтаксис обработки исключений следующий:

```
try
<защищенный блок>
except <фильтр исключений>
<обработчик исключений>
```

Ключевое слово `except` отделяет защищенный блок от блока обработки исключений. *Фильтр исключений* предназначен для

задания значений, определяющих типы исключительных ситуаций, возникновение которых предполагается обрабатывать в блоке исключений.

В отличие от обработчиков завершений фильтры и обработчики исключений выполняются непосредственно операционной системой — нагрузка на компилятор при этом незначительна.

Блок обработки исключений начинает выполняться тогда, когда возникла исключительная ситуация при выполнении защищенного блока. После выполнения блока исключений управление передается на блок, следующий сразу за блоком исключений:

1. Исполняется код перед блоком `try`.

2. Исполняется код внутри блока `try` (защищенный блок). Если возникла исключительная ситуация, то переходим к п. 3, иначе переходим к пункту 4.

3. Исполняется код внутри блока `except` (обработчик исключений).

4. Исполняется код после блока `except`.

Из приведенной последовательности действий видно, что если при исполнении защищенного блока не произошло ошибки, то блок `except` ***не выполняется***. В случае же возникновения ошибки внутри защищенного блока управление передается в обработчик исключений, а после обработки исключительной ситуации ***не возвращается*** в защищенный блок. Поясним это следующими примерами.

Пример 1. Пусть защищенный блок содержит одно действие — присваивает целочисленной переменной *X* значение 0. Тогда предпосылок для возникновения ошибочной ситуации в защищенном блоке нет, и блок исключений никогда не выполняется:

```
try
X = 0
except <фильтр исключений>
Сообщение: "Обработка ошибок"   — этот код никогда не
выполняется
```

Пример 2. Пусть целочисленной переменной *X* присваивается значение 0 перед входом в защищенный блок, а внутри защи-

щенного блока. выполняется последовательность из двух действий: `Y = 10/X` и `X = X+5`. Тогда получаем следующую ситуацию:

```
X = 0
try
Y = 10/X          — возбуждается исключение
X = X+5           — этот код никогда не выполняется
except <фильтр исключений>
Сообщение: "Деление на 0"      — обработка исключения
```

При программировании обработки исключений необходимо помнить, что за блоком `try` всегда должен следовать либо блок `finally`, либо блок `except`. Для одного блока `try` *нельзя* определить одновременно и блок `finally`, и блок `except`. Нельзя также и указать несколько блоков `finally` или `except` для одного блока `try`. Однако блоки `try-except` и `try-finally` могут вкладываться друг в друга, т. е. возможны следующие способы использования:

```
try
...
   try
...
except <фильтр исключений>
...
  finally
  ...
```

или

```
try
...
finally
...
  try
...
   except <фильтр исключений>
   ...
```

1.3. Типы данных

Данные в языках программирования можно классифицировать в зависимости от допустимых операций и целей использования. Такая классификация упрощает понимание программы и описываемого ею процесса обработки данных и позволяет обна-

ружить ошибочное использование данных до выполнения программы (во время компиляции). Отдельные классы в такой классификации называют *типами данных*.

Для всех переменных в языках программирования тип данных задается путем описания, а с каждой операцией однозначно связывается тип ее операндов и результата. Это позволяет однозначно вычислить тип данных любого выражения, что обеспечивает:

1. Возможность определения объема памяти, необходимого для размещения переменных, еще до выполнения обработки и осуществления эффективного управления памятью. Например, описание

```
var Ar_int: array[1 .. 10] of integer
```

определяет переменную `Ar_int` как массив, состоящий из 10 элементов данных типа `integer`. Анализ только текста программы позволяет определить размер области памяти для хранения значений этой переменной.

2. Возможность обнаружения еще до выполнения программы ошибочных операций и выражений. Это позволяет сократить время отладки программы и проверить структуру программы. Например, если переменные `v1` и `v2` описать как

```
var  v1: real;
     v2: integer;
```

то можно выявить следующее ошибочное выражение:

```
v2: = v1 mod 2;
```

Здесь для переменной `v1` типа `real` ошибочно задается операция `mod`, допустимая только для данных типа `integer`.

3. Возможность эффективного анализа семантики и комментирования программ.

Таким образом, тип данных можно определить путем задания множества значений данных, принадлежащих данному типу, и операций, определенных для данного множества.

В [25] рассматривается следующее формализованное определение.

Пусть D — множество, состоящее из всех значений данных, разрешенных в языке программирования L. Если каждому эле-

менту d множества D соответствует определенный тип данных, то множество D можно представить как

$$D(T) = D_{t_1} \cup D_{t_2} \cup \ldots, \quad D_{t_i} \cap D_{t_j} = \varnothing \text{ при } i \neq j,$$

где $T = \{t_1, t_2, \ldots\}$ — множество имен типов данных, разрешенных в языке L, а D_{t_i} есть множество данных типа t_i.

Пусть F — множество операций языка L. Каждый элемент f множества F, как правило, является функцией вида

$$\mathrm{f}: D_{k_1} \times D_{k_2} \times \ldots \times D_{k_n} \rightarrow D_{r_1} \times D_{r_2} \times \ldots \times D_{r_m},$$

где $k_i \in T$, $i = \overline{1..n}$, $r_j \in T$, $j = \overline{1..m}$.

Например, для операции `mod` справедлива следующая функция:

$$mod: D_{integer} \times D_{integer} \rightarrow D_{integer},$$

т. е. и аргументами, и результатом операции `mod` могут являться данные только типа `integer` (целые числа).

Элементами множества F являются:

1. Операции манипулирования данными (арифметические, логические), например вышеуказанная операция mod.

2. Операции отношения, например

$$equal: D_t \times D_t \rightarrow D_{boolean}$$

операция, которая используется для суждения о равенстве двух значений (в большинстве случаев equal(x, y) записывается как $x = y$).

3. Операции преобразования типа, используемые для преобразования типа данных, например, операция преобразования вещественного типа в целый в языке Pascal:

$$round: D_{real} \rightarrow D_{integer}$$ — округление до целого.

4. Операции, обеспечивающие структурирование данных и выборку данных при их реструктурировании.

С использованием приведенной формализации тип данных в языке программирования определяется совокупностью $D = \{D_t : t \in T\}$ и F, которую формально можно записать как $<D, F>$.

Таким образом, чтобы однозначно определить тип данных (а, следовательно, и семантику программы), необходимо однозначно задать множества D и F. Обычно в традиционных языках

программирования это осуществляется неявно, на основании таких общеизвестных понятий, как последовательность и множество, и с учетом организации памяти и операционной системы ЭВМ, используемой для выполнения программ.

Существуют различные языки программирования, и их структура не является единообразной. Разделяют языки процедурного и непроцедурного типа. Среди языков непроцедурного типа выделяют языки функциональные и логические (см. классификацию ЯП). Существует еще целый ряд языков, обладающих своими характерными особенностями, но если рассматривать их с точки зрения типов данных, то между ними есть много общего. Поэтому ниже будут рассмотрены главным образом языки процедурного типа. Основная цель состоит не в рассмотрении типов данных некоторого языка, а в разъяснении понятий, общих для большого числа языков программирования, а приведенные ниже примеры фрагментов программ взяты только для пояснения излагаемого материала.

Исходя из опыта использования языков программирования, некоторые типы данных рассматриваются в качестве основных (рис. 1.6).

Рис. 1.6. Основные типы данных

Основные типы данных классифицируются следующим образом:

1. Простые типы данных.
2. Структурированные типы данных.
3. Ссылочный тип данных.

В языках процедурного типа для присваивания некоторой переменной вновь вычисленного значения служит *оператор присваивания*. Переменная представляет собой своеобразный «ящик»,

в который можно помещать значения данных. Тип переменной явно описывается в тексте программы и определяет множество значений, которое может принимать переменная. Здесь имеют место два случая:

1) каждое значение, присваиваемое переменной, принадлежит определенному типу;

2) по ходу выполнения программы переменной можно присваивать значения любого типа.

Языки программирования, в которых выполняется ограничение первого случая, называются *типизированными*, а во втором случае говорят о *нетипизированных* языках.

В типизированном языке для переменных должны быть явно объявлены их типы. Например, в языке Pascal для описания типа переменной используется следующая конструкция:

```
var x: t;
```

где `x` — имя переменной, а `t` является либо именем типа, либо типовым выражением, состоящим из имен и операторов конструирования структурированного типа данных. При этом именем типа может быть:

- ключевое слово, соответствующее встроенному в данный язык программирования типу данных, например `integer`, `real`;
- идентификатор, задаваемый пользователем и соответствующий определяемому им типу данных.

Рассмотрим, например, следующие описания типов данных `a1`, `a2`:

```
type a1 = array [1 .. 100] of integer;
     a2 = array [1 .. 100] of integer;
```

Эти два описания определяют, по существу, одинаковые массивы целых чисел, состоящие из 100 элементов, которым присвоены различные имена `a1` и `a2`. В данном случае следует рассмотреть две ситуации:

1) `a1` и `a2` имеют одинаковые структуры и являются эквивалентными типами данных;

2) `a1` и `a2` имеют различные имена и являются различными типами данных.

В первом случае считается, что оба типа данных являются *структурно эквивалентными*, несмотря на различие присвоенных им имен, а также имен элементов, образующих эти структуры.

Например, если определить типы данных b и a3:

```
type b = integer;
     a3 = array [1 .. 10] of b
```

то a1, a2 и a3 являются структурно эквивалентными.

Однако существует точка зрения, что имя, присваиваемое типу данных, является также частью типа данных, и при установлении эквивалентности типов данных должны учитываться их имена.

Данная точка зрения соответствует эквивалентности по имени и означает, что типы, которым в программе даны разные имена, считаются разными независимо от того, каковы их структуры, множества значений и т. п. Выбор той или иной точки зрения зависит от языка программирования.

1.3.1. Простые типы данных

Простые типы данных (синонимы — *примитивные*, *элементарные* или *базовые*) не обладают внутренней структурой и служат для объявления только одного значения или задания базовых элементов структурированных типов данных. Набор простых типов данных конечен и включает следующие типы:

1) целый;
2) вещественный;
3) символьный;
4) логический;
5) перечисляемый;
6) интервальный.

Типы *логический*, *символьный*, *перечисляемый*, *интервальный* и *целый* относятся к типам, задаваемым перечислением. Эти типы данных характеризуются тем, что позволяют обозначать данные в виде списка символических имен. Вещественный тип в существующих языках программирования описывает данные, которые не являются вещественными с математической точки зрения, и представляет собой на самом деле конечное множество значений. Хотя по замыслу он близок к вещественному, объективный смысл в перечислении его элементов отсутствует, поэтому обычно вещественный тип не является типом, задаваемым перечислением.

Если T — один из типов, задаваемых перечислением, то обычно упорядоченность элементов во множестве данных D_T задается порядком их перечисления. Упорядоченность последовательности

$$d_1, d_2, ..., d_N$$

элементов множества D_T позволяет ввести функции $succ(d_i)$ и $pred(d_i)$ получения следующего (d_{i+1})и предыдущего (d_{i-1}) элементов по отношению к элементу d_i:

$$succ, pred : D_T \to D_T;$$

$$succ(d_i) = d_{i+1},\ 1 \le i \le N - 1;$$

$$pred(d_i) = d_{i-1},\ 2 \le i \le N.$$

Операция отношения

$$<: D_T \times D_T \to D_{boolean},$$

задающая отношение следования элементов D_T, определяется следующим образом:

$$d_i < d_j \Leftrightarrow i < j.$$

Знак «<» в выражении $i < j$ обозначает отношение «меньше», заданное на множестве целых чисел. В выражении $d_i < d_j$ знак «<» имеет другой смысл и задает порядок следования элементов.

Логический тип данных

Логический тип данных представляет собой множество данных, состоящее из перечисления логических значений `false` и `true`. Логический тип данных именуется как `boolean`, т. е.

$D_{boolean} = \{false, true\}$

и `false < true`.

В качестве операций над логическим типом используются

$and, or : D_{boolean} \times D_{boolean} \to D_{boolean}$;

$not : D_{boolean} \to D_{boolean}$.

Они обозначают, соответственно, конъюнкцию, дизъюнкцию и отрицание (см. п. 1.1).

Данные логического типа чаще всего применяются для формирования значений логических условий.

Символьный тип данных

Символьный тип предназначен для образования текстовых строк, состоит из символов и чаще всего обозначается `char`. Множество данных типа составляют цифры, буквы и специальные знаки. Полный допустимый набор символов определяется языком программирования. Существуют стандарты на символьные множества, например ASCII, ISO, Unicod.

В некоторых языках программирования существуют две стандартные функции `ord` и `chr`, называемые *функциями преобразования*, которые позволяют отображать множество символов на подмножество натуральных чисел и наоборот:

$ord: D_{char} \rightarrow D_{integer}$;

$chr: D_{integer} \rightarrow D_{char}$.

Функция `ord(c)` определяет порядковый номер символа `c` из упорядоченного набора символов, заданных множеством D_{char}, а `chr(i)`, наоборот, определяет символ, порядковый номер которого равен `i`. При этом справедливы следующие соотношения:

```
ord(chr(i)) = i
chr(ord(c)) = c
```

Символьный тип данных впервые (как нововведение) появился в языке Cobol, затем — в языках PL/1, Pascal и пр.

Целый и вещественный типы данных

Целый и вещественный типы данных предназначены для представления числовых значений. Целый и вещественный типы именуются `integer` и `real` соответственно.

Целый тип формируется из множества целочисленных значений, определяемого конкретным языком программирования. Максимальное и минимальное представимые значения зависят от реализации языка и используемой ЭВМ. Диапазон представимых значений для ЭВМ с длиной машинного слова в L бит определяется следующим соотношением:

$$-1^{L-1} \le n \le 2^{L-1} - 1.$$

Вещественный тип представляется в ЭВМ числами с плавающей точкой вида

$$\pm 2^{l} \sum_{i=1}^{N} a_i 2^{-i},$$

где l — целое число, $a_i = 0$ или 1.

Следует отметить, что операции, определенные над данными вещественного типа, в строгом математическом смысле не эквивалентны операциям над вещественными числами, так как при выполнении операций с данными вещественного типа могут возникать ошибки округления результатов. Например, если x, y, z объявлены как данные вещественного типа, то соотношение (x + y) + z = x + (y + z) является некорректным, поэтому условный оператор

```
if x = y then z := 1 else z :=2
```

необходимо преобразовать следующим образом (предполагая, что число epsilon достаточно малое):

```
if abs (x – y)< epsilon then z := 1 else z : = 2
```

Перечисляемый тип данных

Перечисляемый тип относится к задаваемому пользователем типу данных. Задание типа осуществляется путем непосредственного *перечисления* принадлежащих этому типу значений данных:

```
type T = (c1, c2, ..., cn)
```

Здесь c1, c2, ..., cn являются значениями перечисляемого типа T. Например:

```
type day = (Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday);
type shape = (triangle, circle, rectangle);
type color = (black, green, red, yellow);
```

В этом примере множество значений типа данных day состоит из семи элементов (Sunday, ..., Saturday), а множества значений типов shape и color – соответственно, из трех и четырех элементов. Эти элементы не имеют какого-либо смысла — они просто задаются своими идентификаторами.

Для данных такого типа функции pred и succ определяют предыдущие и последующие элементы в соответствии с заданным перечислением порядком:

```
pred(green) = black;
succ(triangle) = circle;
```

Интервальный тип данных

Определение *интервального* типа можно рассматривать как обозначение интервала значений любого заранее определенного перечисляемого типа. Значения элементов такого типа должны находиться внутри заданных границ интервала. Интервальный тип задается указанием нижнего и верхнего значений в этом интервале. В языке Pascal синтаксис задания типа следующий:

```
type T = tmin .. tmax;
```

Например:

```
type digit = 0 .. 9;
type year= 2000 .. 2099;
type Weekday = Monday .. Saturday;
type alphabet = 'A' .. 'Z';
```

В приведенном примере `digit` представляет тип данных, множество значений которого состоит из 10 целых чисел от 0 до 9. `Year` определяет подмножество целых чисел, состоящее из 100 элементов в диапазоне от 2000 до 2099. В данном случае базовым типом является целый тип `integer`. Для `Weekday` в качестве базового типа взят перечисляемый тип `day`, рассмотренный выше. `Weekday` задает тип данных, где последовательно перечислены шесть элементов `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`. Множество значений интервального типа `alphabet` состоит только из символов латинского алфавита (символьные константы взяты в кавычки).

При присваивании значений переменным интервального типа требуется, чтобы значения принадлежали заданному интервалу. Благодаря этому перед выполнением программы можно обнаружить присваивание недопустимых значений. Например, для переменной `x` объявленного выше типа `year` недопустим оператор присваивания:

```
x := 3050
```

1.3.2. Структурированные типы данных

Структурированные типы предназначены для конструирования из конечного набора базовых типов сложных структур данных. Элементы, составляющие структуру, в свою очередь тоже могут обладать структурой.

Основой для создания таких типов являются определенные *правила (операции) конструирования (конструкторы типов)*, причем каждому из них соответствует свой определенный структурированный тип данных. Рассматривают следующие основные структурированные типы данных:

1. Функция с конечной областью определения (во многих языках программирования функцию с конечной областью определения относят к данным типа *массив*).

2. Декартово (прямое) произведение (прямое произведение часто называют *записью* или *структурой*).

3. Объединение (или запись с вариантами).

4. Множество.

При создании структурированных типов данных обычно используются следующие основные операторы:

1) оператор конструирования, который позволяет из элементов образовывать структурированный тип;

2) оператор выбора, позволяющий выделять из структурированного типа данных образующие его элементы.

Пусть C — оператор конструирования, позволяющий получать из элементов с типами t_1, ..., t_n структурированный тип данных t, а S_i — оператор выбора, выделяющий элементов структурированного типа данных с номером i. Тогда C и S_i являются следующими функциями:

$$C: D_{t_1} \times D_{t_2} \times \ldots \times D_{t_n} \to D_t;$$

$$S_i: D_t \to D_{t_i} \quad (i = \overline{1, n}).$$

Для C и S_i справедливы следующие выражения:

1) для любых $d_1 \in D_{t_1}, \ldots, d_n \in D_{t_n}$: $S_i(C(d_1, \ldots, d_n)) = d_i$;

2) для любого $d \in D_t$: $C(S_1(d), \ldots, S_n(d)) = d$.

Конкретный вид операторов C и S_i зависит от самого структурированного типа данных.

Функция с конечной областью определения

Функция с конечной областью определения представляет собой отображение некоторого конечного множества данных на множество данных другого типа. В традиционных языках программирования этот тип называют *массивом*. Конечное множество, являющееся областью определения данной функции (масси-

ва), называется *индексным множеством (типом)*, а его элементы — *индексами*. Значение данной функции для некоторого значения индекса i называется *элементом массива, соответствующим* i, т. е. массив A, элементы которого имеют тип T_0, представляет собой отображение D_I на D_{T_0}:

$$A : D_I \rightarrow D_{T_0}.$$

Массив типа T, таким образом, есть множество элементов данного отображения. Например, в языке Pascal массив с типом индексов `I` и типом элементов `T0` определяется следующим описанием:

```
type T = array [I] of T0
```

Множество индексов I массива должно быть конечным и перечисляемым, поэтому требуется, чтобы тип индекса являлся простым типом с возможностью перечисления, например интервальным, логическим, символьным, перечисляемым. Массив A типа T при $I = \{i_1, i_2, ..., i_n\}$ можно проиллюстрировать следующим образом:

$A[i_1]$	$A[i_2]$	...	$A[i_n]$
i_1	i_2	...	i_n

где $A[i_k]$ является значением A для $I_k \in I$, например:

```
type line = array [1 .. 100] of char;
     shape = (triangle, circle, ellipse, rectangle);
     angle = array [shape] of integer;
```

Массив представляет собой некоторое количество расположенных в определенном порядке элементов одного типа. Индекс предназначен для обеспечения возможности указания на элементы массива.

В приведенном примере последовательность из 100 символов определяет строку (`line`) и каждый из образующих строку символов имеет определенное положение в строке:

C	T	P	O	K	A	.....	!
1	2	3	4	5	6	.....	100

Массив объявленного типа `angle`, например, может соответствовать количеству углов геометрических фигур. В данном слу-

чае типом индекса является перечисляемый тип `shape`, образованный из имен фигур. Сам массив представляет собой вектор-строку целых чисел. Отличие от `line` состоит в том, что для указания элементов массива (числа углов) в качестве индексов использованы идентификаторы перечисляемого типа:

3	0	0	4
triangle	circle	ellipse	rectangle

Необходимо отметить, что, например, значение `triangle` не является отличительной чертой данных, а представляет собой одно из значений элементов, образующих тип данных индексного множества (`shape`).

Операции над данными типа массив. Типичными операциями над данными типа массив являются:

1) задание начальных значений элементов массива;

2) выбор элементов массива по заданным значениям индексов;

3) избирательное обновление массива.

Начальные значения элементов массива A можно идентифицировать указанием значений их индексов. Множество индексов $I = \{i_1, i_2, ..., i_n\}$ является перечисляемым, поэтому A можно задать перечислением значений A для $i \in I$. Если $a_k = A[i_k]$, $k = \overline{1,n}$, то массив A типа T можно представить в виде

$$T(a_1, a_2, ..., a_n).$$

Например, если A — массив типа `angle`, то начальные значения можно задать следующим образом:

```
angle(3, 0, 0, 4)
```

Выбор элементов массива по заданным значениям индексов является операцией, обратной заданию значений элементов массива. Для заданного массива A операция выбора элемента массива заключается в определении значения элемента массива A, соответствующего значению i-го элемента множества индексов I, и обычно выражается с помощью нотации $A[i]$. Например, значение элемента массива A типа `angle`, соответствующее значению индекса `circle`, задается следующим образом:

```
A[circle].
```

Избирательное обновление массива заключается в том, что значение некоторого элемента исходного массива заменяется другим значением. Обычно в языках процедурного типа переменные и присваиваемые им значения имеют одинаковый тип. В процессе выполнения программы значения, присваиваемые переменным, изменяются. Избирательное обновление массива можно реализовать с помощью операции присваивания значений элементам массива:

```
A[i]:=b;
```

Однако при этом необходимо учитывать, что операция присваивания является средством изменения хранимого значения. В языках программирования функционального и логического типа, где отсутствует понятие присваивания значений переменным, избирательное обновление массивов все-таки присутствует как операция над всем массивом.

Многомерные массивы. До сих пор рассматривались одномерные массивы, содержащие только одно индексное множество и базовый тип данных T_0. Аналогичным образом можно рассмотреть многомерные массивы. Многомерный массив представляет собой массив, у которого индексное множество является прямым (декартовым) произведением множеств I_1, I_2, ..., I_n и определяется следующим образом:

```
type T = array [I1, I2, ..., In] of T0
```

Если A — массив указанного типа T, то A является функцией следующего вида:

$$A : D_{I_1} \times D_{I_2} \times \ldots \times D_{I_n} \rightarrow D_{T_0}.$$

В языках программирования принята точка зрения, в соответствии с которой многомерный массив представляет собой структуру, составленную из одномерных массивов, т. е. элементами массива являются массивы с элементами типа массив и т. д. В соответствии с таким представлением рассмотренный выше тип данных T можно описать следующим способом:

```
type T = array[I1] of array [I2] of ... of array [In] of T0
```

Эти два способа описания, строго говоря, различаются. В первом случае значения элементов массива $A[i_1, i_2, ..., i_n]$ определяются только для набора значений индексов $i_1 \in I_1$, $i_2 \in I_2$, ...,

$i_n \in I_n$ и значение $A[i_1]$, соответствующее только значению индекса i_1, не имеет смысла. Во втором же случае значение $A[i_1]$ имеет смысл.

Прямое (декартово) произведение

Прямое (декартово) произведение, как и массив, является одним из основных структурированных типов данных. Его называют также *записью* или *структурой*. В отличие от массива элементы прямого (декартова) произведения могут иметь различные типы.

Множество значений, определенное таким типом, состоит из всех возможных комбинаций значений, относящихся к каждому из множеств значений элементов структуры. Таким образом, число таких комбинаций равно произведению числа элементов в каждом из составляющих множеств.

Тип прямого произведения, состоящий из базовых типов T_1, T_2, ..., T_n, определяется следующим образом:

```
type T= record s1:T1;
              s2:T2;
             .......
              sn:Tn
end
```

Здесь s_1, s_2, ..., s_n — имена элементов, а T_1, T_2, ..., T_n — соответственно, типы данных элементов. Таким образом, множество данных типа T можно представить с помощью прямого (декартова) произведения множеств D_{T_1}, D_{T_2}, ..., D_{T_n}:

$$D_T = D_{T_1} \times D_{T_2} \times \ldots \times D_{T_n}.$$

Каждое отдельное значение D_T можно проиллюстрировать следующим образом:

s_1	d_1
s_2	d_2
	⋮
s_n	d_n

где $d_i \in D_{T_i}$.

Идентификаторы s_1, s_2, ..., s_n используются для указания каждого элемента типа d и часто называются *полями*.

Например, представим объявление записей, принятое в языке Pascal:

```
type
TComplex = record
    re: Real;
    im: Real;
    end;
TDateRec = record
    Day:  1 .. 31;
    Month: 1 .. 12;
    Year: integer;
    end;
TPerson = record
    Family, FirstName, SecondName:string[20];
    Date: TDateRec;
    Year: integer;
    Num:  string[6];
    end;
```

На рис. 1.7 изображены следующие переменные:

- z типа TComplex — объект, описывающий комплексное число, с полями re и im типа Real;
- d типа TDateRec — объект, описывающий дату, с полями Day, Month и Year со значениями из подмножеств типа Integer (например, Day может быть целым числом в интервале от 1 до 31);
- p типа TPerson — объект, содержащий информацию, например, о студенте, с полями Family, Firstname, Secondname, Date, Year, Num.

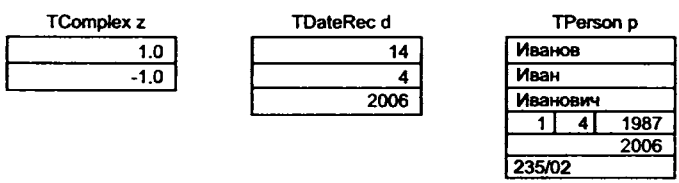


Рис. 1.7. Графическое представление записей

Операции над данными типа запись. Для записей так же, как для массивов, определены следующие основные операции:

1) задание начальных значений элементов записи;
2) выборка элементов записи;
3) избирательное обновление элементов записи.

Если $d_i \in D_{T_i}$ является значением данных типа T_i, то значение типа записи T можно выразить следующим образом:

```
T(d1, d2, ..., dn);
```

Например, переменная тип `TDateRec` может иметь следующие значения полей:

```
TDateRec (14, 4, 2006),
```

Выборка значения заданного элемента (поля) записи осуществляется с помощью операции выборки, которая вводится путем указания имени переменной типа запись и имени поля, разделенным точкой. Значение элемента `si` в значении `r` записи типа `T` в общем случае указывается как `r.si`, например, для переменной `Person` типа `TPerson`:

`Person.FirstName` — значение поля «Имя»;

`Person.Date` — значение поля «Дата рождения».

Избирательное обновление элементов, т. е. замену одного значения элемента в значении записи типа `T` на другое, можно выразить как

```
r.si := b;
```

Например, доступ к элементам записи для переменной `Person` типа `TPerson`:

```
Person.FirstName:='Victor';
Person.Year := 2006;
```

Объединение

Объединение представляет собой множество, отдельные элементы которого классифицируются по категориям. Например, множество геометрических фигур представляет собой объединение категорий (прямоугольник, треугольник, окружность и т. д.), каждая из которых определяется своим набором параметров построения: прямоугольник — высотой и шириной, треугольник — двумя сторонами и углом между ними, окружность — радиусом. Однако, при решении некоторых задач удобно все эти разнородные категории объединять для обработки в одно множество.

Размеченным объединением называется объединение в одно целое при сохранении индивидуальности каждой категории.

Объединение типов данных T_1, T_2 можно получить путем объединения соответствующих множеств данных D_{T_1}, D_{T_2} и определения допустимых операций. Если t_1, t_2 — признаки принадлежности к типам T_1 и T_2, то объединение T типов T_1 и T_2 задается следующим образом:

```
type T    = union  t1:T1;
                   t2:T2;
                   ......
     end;
```

Для множества геометрических фигур определим отдельные категории для хранения информации о прямоугольниках, треугольниках и окружностях соответственно:

```
type RFigure = record
                  RHeight, RWidth: real
               end;
type TFigure = record
                  TSize1, TSize2, Angle: real
               end;
type CFigure = record
                  Radius: real
               end;
```

Тогда можно определить тип данных `Figure` как прямую сумму `RFigure`, `TFigure` и `CFigure`:

```
type Figure = union
                  Rectangle: RFigure;
                  Triangle: TFigure;
                  Circle: CFigure;
           end;
```

В некоторых ЯП высокого уровня размеченное объединение представляют как *запись с вариантами*:

```
TKind = (Rectangle, Triangle, Circle);
Figure = record
       case Kind: TKind of
       Rectangle: (RHeight, RWidth: Real);
       Triangle: (TSize1, TSize2, Angle: Real);
       Circle: (Radius: Real);
       end;
```

Множество данных типа объединения T можно задать с помощью следующего выражения:

$$D_t = \{t_1 : D_1 | d_1 \in T_{t_1}\} \cup \{t_2 : D_2 | d_2 \in T_{t_2}\} \cup \ldots .$$

Таким образом, значение типа объединения $t_i : d_i$, образуется из признака t_i, определяющего элемент, и значения d_i:

t_i	Признак
d_i	Значение

Например, значениями типа `Figure` могут быть следующие (рис. 1.8).

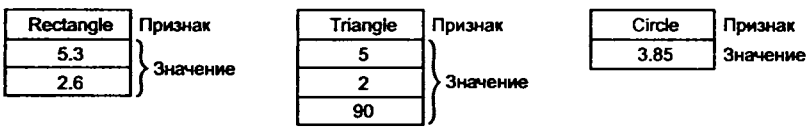


Рис. 1.8. Пример размеченного объединения

Операции над данными типа объединения. Основными операциями над данными такого типа являются:

1) операция задания объединения;

2) операция преобразования объединения к значениям составляющих.

Значение d типа объединения T означает, что значениям составляющих типа T_1, T_2, ... поставлены в соответствие признаки t_1, t_2, ..., и это выражается следующим образом:

$$t_i : d_i, \text{ где } d_i \in D_{T_i} \quad i = 1, 2, \ldots .$$

В данном случае t_i можно рассматривать как функцию вида

$$t_i : D_{T_i} \rightarrow D_T.$$

Например, значения типа `Figure` выражаются следующим образом:

```
Rectangle: RFigure (5.3,2.6);
Triangle: TFigure (5, 2, 90);
Circle: CFigure (3.85);
```

Преобразование объединения к значениям составляющих подчиняется следующему правилу: пусть d — значение типа прямой суммы T. Для d имеет место одна из форм $t_1 : d_1$, где $d_1 \in D_{T_1}$, или $t_2 : d_2$, $d_2 \in D_{T_2}$. При $d = t_1 : d_1$ запись называют составляющей t_1 значения d и записывают:

```
d1 = d.t1.
```

Множество

Множество представляет собой тип данных, значениями которого является заданное подмножество значений базового типа. В качестве базового типа выступает порядковый тип.

Множество, состоящее из перечисления элементов базового типа T0, определяется следующим образом:

```
type T = set of T0 ;
```

Например, тип, определяющий множество цифр:

```
type Digit = set of 0..9;
```

Основными операциями над данными типа множество являются:

1) задание значений элементов множества;

2) объединение, пересечение и разность множеств;

3) определение принадлежности множеству.

Множество, элементами которого являются *a*, *b*, ..., *c*, принадлежащие базовому типу, можно представить следующим образом:

```
T[a, b, ..., c];
```

Если базовый тип представляет собой интервал, то получаем выражение

```
T[a .. c];
```

Операции над данными типа множество. Для заданных множеств S_1, $S_2 \in D_T$ операции объединения, пересечения и разности

$$\cup, \cap, -: D_T \times D_T \to D_T$$

определены следующим образом:

$$S_1 \cup S_2 = \{d \mid d \in S_1 \text{ or } d \in S_2\};$$

$$S_1 \cap S_2 = \{d \mid d \in S_1 \text{ and } d \in S_2\};$$

$$S_1 - S_2 = \{d \mid d \in S_1 \text{ and } d \notin S_2\}.$$

Операция, определяющая принадлежность элемента множеству, задается следующим образом:

$$in : D_T \times D_T \rightarrow D_{boolean}.$$

$$\text{Для } d \in D_T,\ S \in D_T \quad d \ in \ S = \begin{cases} true, & d \in S; \\ false, & d \notin S. \end{cases}$$

1.3.3. Ссылочный тип данных

Ссылочный тип данных является средством организации и обработки сложных изменяющихся структур данных. Этот тип данных предназначен для обеспечения возможности указания на данные других типов и называется *указателем (ссылкой)*.

Введение указателей характерно для языков процедурного типа, в которых существует понятие области памяти для хранения данных. При правильном использовании указателей достигается большой эффект при выполнении программ. Однако при введении ссылочного типа данных в языки программирования высокого уровня необходимо обращать внимание на правильность их применения.

В языках программирования переменные как имена областей для хранения значений данных играют важную роль с точки зрения описания вычислительного процесса. То есть при выполнении программы по значениям данных, присваиваемых переменным, рассчитываются новые значения, которые вновь присваиваются переменным. Реализуемый программой вычислительный процесс проходит через ряд «состояний», которые определяются значениями содержащихся в программе переменных и структурой управления программы.

Среди всех переменных выделяют переменные, предназначенные для хранения данных *ссылочного типа*. Идентификаторам таких переменных соответствуют не адреса (имена) ячеек памяти, используемые для хранения значений переменных,

Рис. 1.9. Пример переменной ссылочного типа

а ссылки на другие переменные. Тот факт, что значение переменной x представляет собой ссылку на переменную y, можно проиллюстрировать следующим образом — рис. 1.9.

Статические и динамические переменные

Одновременное выделение памяти для всех переменных в ходе выполнения программы необязательно. Обычно при выполнении программы переменные существуют в течение некоторого промежутка времени, а затем уничтожаются. В зависимости от способов создания (размещения) и уничтожения переменные подразделяются на статические и динамические.

Статические переменные представляют собой переменные, которые объявляются в некоторых процедурах или блоках. Такие переменные формируются автоматически при передаче управления процедуре и уничтожаются при выходе из нее. Время существования таких переменных соответствует времени выполнения данной процедуры.

С другой стороны, переменные могут создаваться *динамически* в зависимости от наличия в программе специальных операторов объявления и размещения переменных. Когда необходимость в переменных отпадает, они уничтожаются с помощью оператора уничтожения. Время существования динамических переменных зависит от выполнения специальных операторов.

Описание ссылочного типа выполняется следующим образом:

```
type T = ↑T0,
```

где T_0 — тип переменной (динамической), на которую объявляется ссылка, при этом тип T_0 должен быть описан в программе. Множество D_T данных типа T представляет собой множество, полностью состоящее из ссылочных значений динамических переменных типа T_0.

Объявление и уничтожение динамических переменных

Пусть существует следующее описание переменной p ссылочного типа:

```
var p:T
```

В этом случае значением переменной p является ссылочное значение на переменную типа T_0 (или адрес переменной типа T_0).

Для всех ссылочных типов данных предполагается наличие значения `nil`, которое используется как начальное значение `p` и означает, что память под переменную базового типа еще не была отведена.

Образование динамической переменной, содержащей непосредственно ссылочное значение, осуществляется в результате выполнения специальной процедуры `new(p)`. Процедура `new(p)` обеспечивает:

1) размещение переменной типа `T0` в памяти;

2) присваивание переменной `p` ссылки на размещенную переменную.

Для обращения к значению динамической переменной, образованной с помощью переменной `p`, служит идентификатор `p↑`. Значение, которое имела переменная `p` перед очередным выполнением процедуры `new(p)`, теряется, следовательно, теряется и область памяти, в которой было размещено предыдущее значение переменной типа `T0` (рис. 1.10). Если в дальнейшем на переменную не осуществляется ссылка с помощью какой-либо другой переменной ссылочного типа, то она становится так называемым «мусором». Накопление большого количества такого «мусора» влечет увеличение затрат памяти на его хранение. Чтобы избежать этого, необходимо иметь возможность повторно использовать области памяти, распределенные для хранения динамических переменных, на которые более нет ссылок. Для этой цели возможно использование процедуры `dispose(p)`, обеспечивающей уничтожение значений переменной, на которую в настоящий момент осуществляется ссылка с помощью переменной `p`, или автоматическое обнаружение и уничтожение значений тех переменных, на которые не осуществляется ссылка.

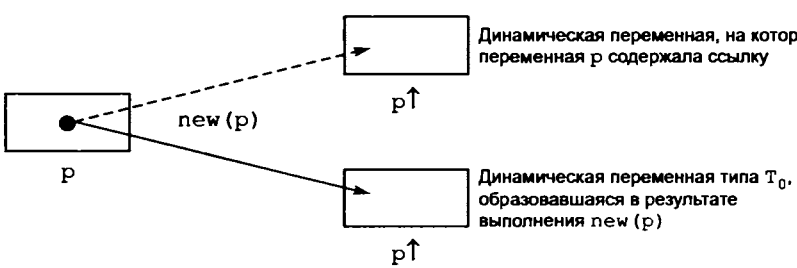


Рис. 1.10. Выполнение процедуры размещения динамических переменных `new(p)`

Этот прием, обеспечивающий возможность повторного использования памяти, называется сборкой «мусора».

Примеры использования ссылочного типа при программировании структур данных приведены в гл. 4.

1.4. Структуры данных

В п. 1.3 были рассмотрены основные структурированные типы, т. е. структуры для размещения данных — массивы, записи, множества. Они названы основными, так как являются основой для построения более сложных структур. В то же время объявление типа данных даже для самой сложной структуры фиксирует множество возможных значений и (почти всегда) размер выделяемой памяти. Однако задачи, в которых точные объемы обрабатываемых данных заранее неизвестны, для своего решения требуют использования более сложных структур, обладающих способностью к изменению в процессе обработки. Такие данные относятся к разряду данных с *динамической структурой*.

Рассмотрим типологию и разновидности таких структур данных с точки зрения особенности их организации. Структуру данных определяют:

- алгоритм выборки отдельных элементов данных;
- особенности организации и обработки информации.

То есть структура данных — это способ отображения значений в памяти: размер области и порядок ее выделения (который и определит характер процедуры адресации/выборки). Зачастую именно успешность структурирования данных определяет сложность процедур их обработки [10].

Классификация структур данных должна проводиться с двух точек зрения.

1. По *характеру взаимосвязи элементов структуры* (с точки зрения порядка их размещения/выборки) виды структур можно разделить на *линейные* и *нелинейные*.

2. По *характеру информации*, представляемой структурой (с точки зрения однородности и «элементарности» типов данных), — на *однородные* структуры, где все элементы имеют один тип данных, и *неоднородные* (композиционные), где элементы относятся к разным типам данных.

1.4.1. Линейные структуры данных

К линейным структурам традиционно относят последовательности[1]. Порядок следования (и, соответственно, выборки) элементов таких структур имеет линейный характер и соответствует порядку их расположения в памяти: один за другим без каких-либо промежутков. Адрес элемента соответствует его положению и определяется индексом, задающим порядковый номер элемента в последовательности размещения. К отдельному элементу имеется прямой доступ, если известен его индекс. Индекс в этом случае позволяет достаточно просто вычислять значение физического адреса элемента по значению его индекса.

Последовательность так же, как и массив, представляет собой совокупность однотипных элементов. Однако число элементов до размещения неизвестно. И хотя каждая конкретная последовательность имеет конечную длину, до начала обработки (и, соответственно, размещения) необходимо считать длину последовательности бесконечной. Принципиальность такого предположения выражается в том, что необходимо предусматривать специальную процедуру использования памяти (выделения/освобождения) и, возможно, алгоритм обработки последовательности по частям. Важность рассмотрения такого типа данных обусловлена тем, что именно он превалирует в операциях ввода-вывода с устройствами внешней памяти. Именно последовательный доступ позволяет организовать «потоковые» операции: однородность позволяет рассматривать пересылаемые данные как непрерывный поток. Поток не может быть прерван по контекстно определяемому условию, например, при пересылке текста — по значению кода «перевод строки», и это не заставляет программу анализировать значение каждого очередного элемента. И, кроме того, последовательный доступ — это простота управления памятью и устройством ввода-вывода.

В зависимости от типа элементов, вида разрешенных операций и способов использования в программах существует не-

---

[1] К линейным структурам данных относят также и массивы. Массив как линейная совокупность однотипных элементов, число которых известно до его размещения, относится к статическим структурам данных и рассмотрен ранее (п. 1.3). В некоторых языках программирования массив задается с помощью стандартного типа данных.

сколько разновидностей последовательности. Типичными примерами последовательностей являются последовательности символов, последовательный файл, очередь и стек. В некоторых ЯП их рассматривают как самостоятельные типы данных, тогда тип T, представляющий собой последовательность элементов типа T_0, определяется следующим образом:

```
type T = sequence of T0;
```

Последовательность T определяет множество конечных последовательностей произвольной длины, состоящих из элементов типа T_0.

Основными операциями над данными типа последовательность являются:

1) формирование последовательности;

2) выборка элементов последовательности;

3) включение-исключение элементов в (из) последовательность (и).

Последовательность типа T, состоящую из элементов e_1, e_2, ..., e_n, можно представить в виде $T(e_1, e_2, ..., e_n)$. В частности, при $n = 0$ последовательность $T(\,)$ называют *пустой*.

Пусть `x` — последовательность, тогда для ее элементов можно определить следующий набор операций:

- `x[i]` — выбор `i`-го элемента `x`;
- `first(x)` — выбор начального элемента `x`;
- `last(x)` — выбор конечного элемента `x`;
- `tail(x)` — исключение начального элемента последовательности `x`;
- `initial(x)` — исключение последнего элемента последовательности `x`;
- `appendl(x,e)` — добавление элемента `e` перед последовательностью `x` (слева);
- `appendr(x,e)` — добавление элемента `e` после последовательности `x` (справа);
- `empty(x)` — определение пустой последовательности (`true`, если `x = T(); false`, если `x` не является пустой).

В зависимости от ограничений на использование представленных операций выделяют несколько разновидностей последовательностей. Далее рассмотрены наиболее важные из них.

Последовательный файл

Последовательным файлом называется последовательность, для которой определены следующие пять операций:

1) формирование пустой последовательности `T()`;
2) `first(x)`;
3) `tail(x)`;
4) `appendr(x,e)`;
5) `empty(x)`.

Последовательный файл представляет собой последовательность элементов, в которой допускается выборка (доступ) начального элемента, а также добавление элемента в конец последовательности. Такие последовательности традиционно реализуются на внешней запоминающей среде последовательного доступа, причем их запись возможна только в одном направлении.

Обработка файла реализуется путем введения файловой переменной `f` и дополнительной переменной типа `T0`, которая называется *буферной переменной файла* и обозначается через `f↑`. Она определяет текущее значение того элемента файла, который является объектом очередной операции обработки. Совокупность значений элементов файла и положение буферной переменной определяют текущее состояние файла (рис. 1.11).

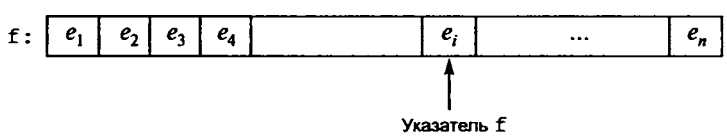


Рис. 1.11. Структура данных последовательный файл

Определены следующие основные операторы, используемые для управления файлом:

`Rewrite(f)` — служит для формирования нового файла и соответствует заданию переменной `f` значения, равного пустому файлу;

`Reset(f)` — обеспечивает установку файла в начальную позицию;

`Write(f,e)` — приписывает в конец файла `f` значение переменной `e`;

`Read(f,e)` — считывает соответствующее положению указателя значение элемента файла `f` в переменную `e`.

`eof(f)` — отражает факт достижения конца файла (логическая функция, в случае достижения конца файла принимает значение `true`, в противном случае ее значение есть `false`).

Стек

Стек — это последовательность, для которой определены следующие операторы:

1) образование пустой последовательности `T()`;
2) `first(x)`;
3) `tail(x)`;
4) `appendl(x, e)`;
5) `empty(x)`.

Стек является наиболее широко используемым типом данных и применяется, например, при анализе языковых конструкций. Для стека добавление и извлечение элементов возможно только с одного конца последовательности — элемент данных, добавленный к последовательности последним, извлекается из нее первым. В этом смысле стековая память является памятью с дисциплиной обслуживания «последним вошел — первым вышел» (Last In First Out — LIFO) (рис. 1.12).

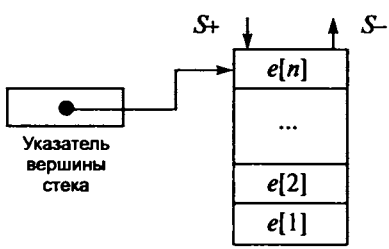


Рис. 1.12. Структура данных стек

Операции `first`, `tail`, `appendl` применительно к стеку обычно называются `top`, `pop`, `push` соответственно. В языках процедурного типа стек реализуется с помощью *стековой переменной*. Данные операции предназначены для манипулирования значениями такой переменной.

Пусть задана стековая переменная `S`, тогда:

- `top(S)` — операция указания на самый верхний элемент `s` (обычно стек изображается вертикально и будем считать, что добавление и извлечение элемента выполняется в его верхней части);

- `pop(S)` — операция извлечения из стека `S` самого верхнего элемента;
- `push(S,e)` — операция добавления элемента `e` к `S`.

`First`, `tail`, `appendl` для левого края последовательности соответствуют операциям `top`, `pop`, `push` для верхней части стека.

Очередь

Очередь — это последовательность, для которой определены следующие операции:

1) образование пустой последовательности `T()`;
2) `last(x)`;
3) `initial(x)`;
4) `appendl(x,e)`.

Добавление элемента к очереди осуществляется с ее левого конца с помощью оператора `appendl`, а извлечение элемента из очереди производится с правого конца с помощью операторов `last` и `initial`. В связи с этим очередь является памятью с дисциплиной обслуживания «первым вошел — первым вышел» (First In First Out — FIFO) (рис. 1.13).

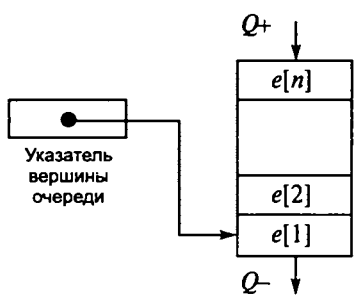


Рис. 1.13. Структура данных очередь

Очереди используются в случае, когда данные обрабатываются в порядке их поступления или образования.

Операции `initial`, `appendl`, `last` применительно к очереди обычно реализуются операторами управления обработкой `enter(x,e)`, `leave(x,e)`:

- `enter(x,e)` — эквивалентно `appendl(x,e)`;
- `leave(x,e)` — эквивалентно `initial(x)` и `last(x)`: удаление последнего элемента последовательности и перемещение его в переменную `e`.

1.4.2. Нелинейные структуры данных

К нелинейным структурам традиционно относят списки, деревья и графы. Порядок следования (и, соответственно, выборки) элементов таких структур может не соответствовать порядку расположения элементов в памяти. Списки представляют собой пример линейного упорядочения, деревья — двумерного, сети — произвольного. Соответственно различаются методы и средства, обеспечивающие последовательность выборки элементов данных.

Списки

Список представляет собой совокупность однотипных элементов. Однако порядок выборки элементов может отличаться от порядка следования в памяти, определенного при размещении. Наиболее очевидный способ установления однонаправленного порядка выборки элементов — это сопоставить каждому элементу списка ссылку, указывающую на следующий элемент. Соответственно, для организации двунаправленного списка, допускающего также выборку в обратном порядке, каждый элемент должен иметь ссылку на предыдущий. Такая организация уже не допускает возможности прямого доступа, например по номеру элемента.

Кроме того, число элементов списка, как и в случае последовательностей, может быть неизвестно до размещения и до начала обработки (и, соответственно, размещения) необходимо считать длину списка бесконечной, что ведет к необходимости предусматривать специальную процедуру выделения/освобождения памяти.

Таким образом, с точки зрения физической реализации элемент списка должен быть *составным*, включающим собственно информативные данные и дополнительные данные (*ссылки*), определяющие порядок доступа к элементам.

В зависимости от способа построения списка и предполагаемых путей доступа к элементам различают следующие виды списков:

- однонаправленные;
- двунаправленные;
- циклические.

Наиболее простым и естественным видом списка является однонаправленный список, в котором каждый элемент содержит обязательно только одну ссылку — на следующий по порядку элемент. Списковая переменная в этом случае содержит начало списка (рис. 1.14, *а*).

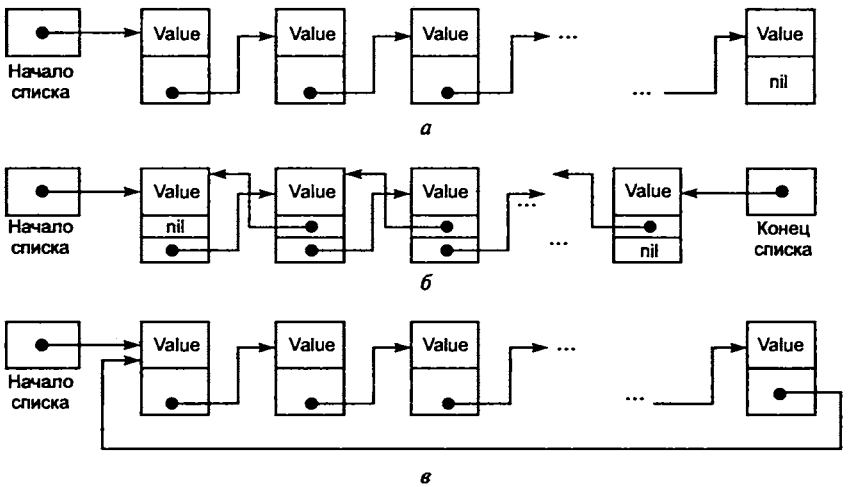


Рис. 1.14. Списковые структуры:
а — однонаправленный список; *б* — двунаправленный список; *в* — циклический однонаправленный список

Однонаправленные списки предусматривают жесткий порядок перебора элементов — только в одном направлении, от первого к последнему. Двунаправленный список представляет собой цепочку элементов, в которой каждый элемент содержит ссылку не только на следующий, но и на предыдущий. Для таких списков нужна обычно дополнительная переменная — указатель на последний элемент списка (рис. 1.14, *б*).

В циклических или кольцевых списках порядок следования элементов зацикливается следующим образом: в однонаправленном кольцевом списке последний элемент ссылается на первый как на следующий, а в двунаправленном кольцевом списке последний ссылается на первый как на следующий, а первый — на последний как на предыдущий (рис. 1.14, *в*).

Добавление и исключение элементов списка можно выполнять с помощью простой операции изменения значения указателя. Например, алгоритм добавления элемента в однонаправлен-

ный список за элементом с номером i состоит из следующей последовательности действий:

1) разместить в памяти новый элемент s с помощью процедуры new(s): в переменной s — ссылка на новый элемент;

2) у элемента s↑ значению ссылки на следующий присвоить значение ссылки на следующий элемента с номером i;

3) значению ссылки на следующий у элемента с номером i поменять на значение переменной s.

Следует обратить внимание на то, что этот алгоритм будет видоизменен в граничных случаях — добавление нового элемента перед первым и после последнего.

Деревья

Дерево представляет собой иерархию элементов, называемых узлами. На самом верхнем уровне иерархии имеется только один узел — корень. Каждый узел, кроме корня, связан с одним узлом на более высоком уровне, называемым исходным узлом для данного узла. Каждый элемент имеет только один исходный. Каждый элемент может быть связан с одним или несколькими элементами на более низком уровне, которые называются порожденными. Элементы, расположенные в конце ветви, т. е. не имеющие порожденных, называются листьями.

Существует несколько способов представления структуры дерева. Например, дерево может быть определено как иерархия узлов, в которой:

1) самый верхний уровень иерархии имеет один узел, называемый *корнем*;

2) все узлы, кроме корня, связываются с одним и только одним узлом на более высоком уровне по отношению к ним самим.

Такое определение в части организации связей совпадает со списком, и, в частности, список представляет вырожденный случай дерева, в котором каждая вершина имеет не более одного поддерева.

В соответствии со структурой заполнения деревья подразделяются на сбалансированные и несбалансированные.

Сбалансированное дерево (в отличие от несбалансированного) в каждом узле имеет одинаковое число ветвей, причем процесс включения новых ветвей в узлы дерева идет сверху вниз, а на каждом уровне дерева — слева направо (рис. 1.15).

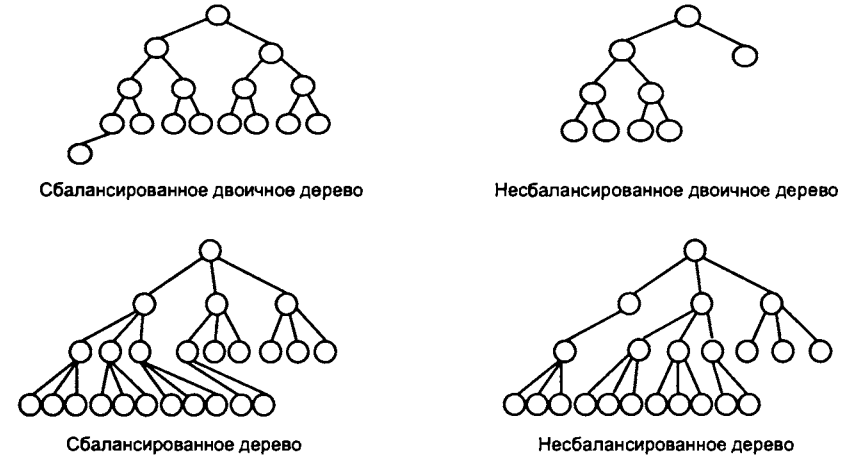


Рис. 1.15. Примеры сбалансированных (*а*) и несбалансированных (*б*) деревьев

Среди древовидных структур выделяют двоичные деревья. *Двоичные деревья* — это особая категория древовидных структур, в которой допускается не более двух ветвей для одного узла.

Любые связи в дереве с любым количеством ветвей можно представить в виде двоичных древовидных структур. Рисунок 1.16 иллюстрирует представление произвольного дерева в виде двоичного дерева.

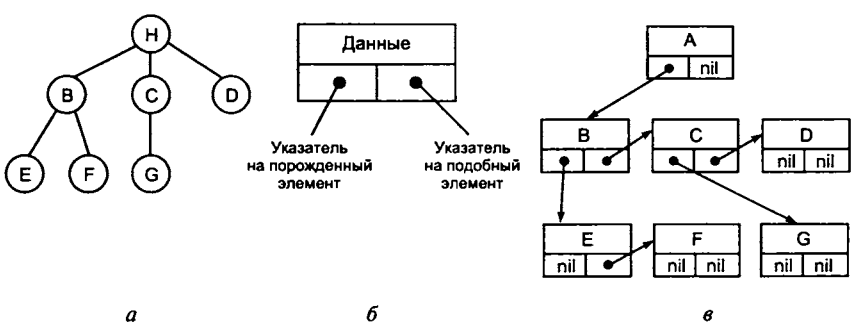


Рис. 1.16. Представление древовидной структуры в виде двоичного дерева со ссылками на подобный и порожденный элементы:
а — дерево; *б* — структура элемента (узла); *в* — структура с указателями

При таком представлении каждый элемент может иметь указатели как на порожденные, так и на подобные элементы.

Графовые структуры

Графовая структура представляет собой наиболее общий (произвольный) случай размещения и связей отдельных элементов в памяти. Рассмотренные выше списковые структуры и деревья можно рассматривать как частные случаи графа.

Существуют различные способы представления графовых структур в памяти ЭВМ. Один из них — представление графа в виде совокупности узлов и дуг. Дуги при этом представляют собой однотипные структуры, состоящие из двух частей: данные и пара указателей, соответственно, на левый и правый узлы. Узлы графа могут иметь структуры, различающиеся по количеству указателей (связей) (рис. 1.17).

Рис. 1.17. Структура представления узлов и дуг графа: *а* — узлы; *б* — дуги

На рис. 1.18 представлен пример реализации структуры для графа, описывающего схему прокладки сетевого кабеля между отделами некоторого учреждения.

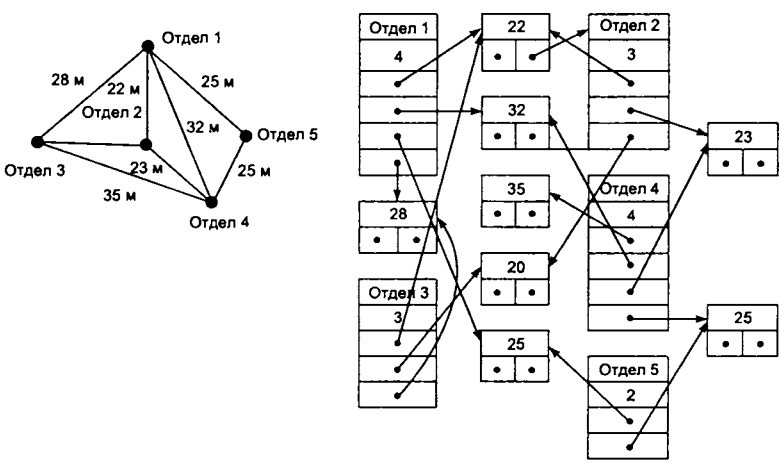


Рис. 1.18. Пример представления графа для схемы сетевого провода

1.5. Процедуры и функции

При разработке программ на алгоритмических языках широко используется понятие *подпрограммы*. Подпрограмма — это средство, позволяющее многократно использовать в разных местах основной программы один раз описанный фрагмент алгоритма. Подпрограмма представляет собой блок, который может вызываться из разных частей программы. При вызове таких блоков в них могут передаваться некоторые переменные, константы, выражения, которые являются параметрами подпрограммы. Подпрограммы в качестве результата могут формировать некоторые значения.

Функциональная декомпозиция программы и выделение подпрограмм позволяет наделить подпрограмму следующими важными отличительными особенностями.

1. Реализация отдельной подпрограммы не зависит от реализации программы в целом (или других подпрограмм), что позволяет впоследствии составлять программу из фрагментов, создаваемых разработчиками, работающими независимо друг от друга.

2. Декомпозиция программы позволяет упростить модификацию программы. Если реализация подпрограммы меняется, но способ ее вызова остается прежним, то внесенные в подпрограмму изменения не повлияют на оставшуюся часть программы. Разумное определение необходимых подпрограмм на начальном этапе разработки с учетом возможных модификаций может значительно сократить объем работ.

Для понимания и сопровождения программы, использующей подпрограммы, важно знать, что реализуют собой сами используемые подпрограммы. Таким образом, при разработке программы важную роль играет ее спецификация, т. е. описание того, что делает подпрограмма, какие параметры (и какого типа) могут быть заданы и какой результат будет получен. Спецификация программы в дальнейшем позволяет ее использовать, не заботясь о том, каким образом будет получен результат. Тем самым абстракция на уровне подпрограмм позволяет расширить заданную некоторым языком программирования виртуальную машину новой операцией.

В большинстве ЯП не проводится концептуального различия между такими объектами, как программа и подпрограмма (про-

цедура, функция). В связи с этим всякая подпрограмма может приобретать иерархическую структуру и включать подчиненные (вызываемые) подпрограммы, процедуры и т. д.

1.5.1. Объявление подпрограмм

Во многих языках программирования существуют две разновидности подпрограмм — процедуры и функции. Каждое объявление процедуры или функции содержит обязательный заголовок, за которым следуют разделы локальных объявлений (аналогичных разделам объявлений программы) и составной оператор (блок), реализующий алгоритм подпрограммы.

Вызов процедуры на исполнение активизируется с помощью оператора процедуры. Функция активизируется при вычислении выражения, содержащего вызов функции, а возвращаемое функцией значение подставляется в это выражение.

Объявление процедур

Синтаксис объявления процедуры (например, в языке программирования Pascal) следующий:

```
procedure <Имя процедуры> (<Список формальных параметров>);
  <Раздел локальных объявлений>
  <Составной оператор>
```

В заголовке процедуры указывается имя процедуры и описывается список формальных параметров (если он присутствует).

За заголовком может следовать раздел локальных объявлений (меток, типов, констант, переменных, вложенных процедур и функций). Раздел локальных объявлений может отсутствовать в том случае, если процедура использует только глобальные объявления.

Запуск процедуры на исполнение осуществляется с помощью оператора процедуры, в котором содержатся имя процедуры и фактические параметры.

Последовательность операторов, реализующих алгоритм процедуры, записывается внутри составного оператора (блока) процедуры. Если в каком-либо операторе внутри блока процедуры используется идентификатор самой процедуры, то процедура будет выполняться *рекурсивно*, т. е. при выполнении будет обращаться сама к себе (см. п. 1.5).

Приведем пример объявления процедуры в языке Pascal:

```
// Преобразование целого числа в строку восьмеричного
// представления
procedure OctString(Nmb: Integer; var S: string);
    // Заголовок процедуры со списком формальных параметров:
    // Nmb – исходное целое число; S – строка для записи
    // результата преобразования
var
   P: Integer;  // Объявление локальной переменной P
begin
       P := Abs(Nmb);
       S := ''; // "пустая" строка
       repeat
         S := chr(P mod 8) + S;
         P := P div 8;
       until P = 0;
       if Nmb < 0 then
         S := '-' + S;
end;
```

В дальнейшем для вызова процедуры из основной программы или другой подпрограммы необходимо использовать оператор процедуры со списком фактических параметров, которые должны совпадать по количеству и типам с формальными параметрами процедуры. Например, в результате выполнения фрагмента программы:

```
OctString(InpNmb,ResultString);
ResultString := 'Число '+str(InpNmb)+
                ' в восьмеричной с.с. равно '+ResultString;
```

в переменной `ResultString` типа `string` формируется запись старого и нового представления целого числа `InpNmb`.

Объявления функций

Функция — это подпрограмма, вычисляющая и возвращающая некоторое значение. Оператор функции может быть использован в качестве операнда при построении выражения.

Например, синтаксис объявления функции в языке Pascal следующий:

```
function <Имя функции>(<Список формальных параметров>):
                                <Тип результата>;
<Раздел локальных объявлений>;
<Составной оператор>
```

Основное отличие заголовка функции от заголовка процедуры (помимо используемого для объявления служебного слова) в том, что заголовок функции дополнительно содержит указание на тип возвращаемого функцией значения — тип результата.

Оператор функции при ее вызове обычно стоит либо в правой части оператора присваивания, либо входит на правах операнда в выражение, либо указывается в качестве фактического параметра при вызове другой подпрограммы. Вызов функции содержит идентификатор функции и список фактических параметров, совпадающий по размеру и типам со списком формальных параметров. После выполнения тела функции возвращается значение, тип которого совпадает с типом результата функции.

Операторная часть тела функции содержит операторы, реализующие алгоритм получения результата объявленного типа, при этом в операторной части должен находиться по крайней мере один оператор присваивания, в котором в левой части стоит идентификатор функции. Результатом выполнения функции будет последнее значение, присваиваемое идентификатору функции. Если такого оператора присваивания нет или он не выполняется, то возвращаемое функцией значение не будет определено.

Если идентификатор функции используется для вызова функции внутри операторной части тела функции, то функция выполняется рекурсивно.

Приведем пример объявления функции:

```
// Функция вычисления суммы квадратов первых N чисел
// натурального ряда
function SumSqr(N: integer): integer;
var
   S, i: integer;
begin
   S := 1;
   for i := 2 to N do
     S := S+i*i;
   SumSqr := S; // Присваивание значения результату функции
end;
```

Для вызова функции из основной программы или из другой подпрограммы (например, при вычислении значения выражения) необходимо в выражении указать оператор функции со списком фактических параметров:

```
LineLen := Log10(X)/SumSqr(I);
```

Здесь `Log10(X)` — вызов стандартной функции вычисления десятичного логарифма фактического параметра `X`;

`SumSqr(I)` — вызов функции вычисления суммы квадратов с фактическим параметром `I`.

Объявления (типы, переменные, константы), использующиеся любой подпрограммой, относятся к одной из двух категорий — категории *локальных* объявлений и категории *глобальных* объявлений. Локальные объявления принадлежат подпрограмме, описаны внутри нее и могут использоваться только ею. Переменные, объявленные таким образом, формируются автоматически при передаче управления процедуре и уничтожаются при выходе из нее. Время существования таких переменных соответствует времени выполнения данной процедуры, поэтому наиболее оптимальным способом организации памяти для хранения значений таких переменных и другой управляющей информации является использование *стеков* (см. п. 1.3).

Глобальные объявления принадлежат программе в целом и доступны как самой программе, так и всем ее подпрограммам. Обмен данными между основной программой и ее подпрограммами обычно осуществляется посредством глобальных переменных.

Если имя глобального объявления совпадает с именем локального, то внутри подпрограммы объявление обычно интерпретируется как локальное, и все изменения, вносимые, например, в значение такой переменной, актуальны только в рамках подпрограммы.

1.5.2. Параметры подпрограмм

Подпрограмма выполняет преобразование входных параметров в выходные — это есть отображение набора значений входных аргументов в выходной набор результатов с возможной модификацией входных значений.

Формальные и фактические параметры

Объявление подпрограммы может содержать список параметров, которые называются *формальными*.

Наделяя подпрограмму параметрами, абстрагируются от конкретных используемых данных. Эта абстракция определяется в терминах формальных параметров. Фактические данные связы-

ваются с этими параметрами в момент использования подпрограммы. Значения конкретных используемых данных являются несущественными, важно только их количество и тип. Преимущество таких обобщений заключается в том, что они уменьшают объем программ.

Каждый параметр из списка формальных параметров является локальным по отношению к подпрограмме, для которой он объявлен. Это означает, что глобальные переменные, имена которых совпадают с именами формальных параметров, становятся недоступными для использования в подпрограмме.

Все формальные параметры можно разбить на две категории:

- параметры, вызываемые подпрограммой *по своему значению* (т. е. параметры, которые передают в подпрограмму свое значение и не меняются в результате выполнения подпрограммы);
- параметры, вызываемые подпрограммой *по наименованию* (т. е. параметры, которые становятся доступными для изменения внутри подпрограммы).

Главное различие этих двух категорий — в механизме передачи параметров в подпрограмму.

При обращении к подпрограмме формальные параметры заменяются на соответствующие по типу и категории *фактические параметры* вызывающей программы или подпрограммы.

Вызов параметров по значению

При вызове параметра по значению происходит копирование памяти, занимаемой параметром, в стек и использование в дальнейшем в операторах подпрограммы локальной копии параметра. Основное значение параметра (глобальное по отношению к подпрограмме) при этом остается без изменения. Следует отметить, что использование такого механизма при передаче, например, массивов большой длины может отрицательно влиять на быстродействие программы и заполняет стек лишней информацией.

Фактический параметр должен быть совместим по присваиванию с типом формального параметра-значения.

Рассмотрим пример объявления функции с параметром-значением:

```
// Функция вычисления количества запятых в строке
function NmbComma( S: string ): integer;
var i, L, Nmb :integer;
```

```
begin
   Nmb := 0;
   L := Length(S);
   for i := 1 to L do
     if S[i] = ?,? then inc(Nmb);
   NmbComma := Nmb;
end;
```

Вызов параметров по наименованию

При вызове параметра по наименованию в подпрограмму передается адрес памяти (глобальной по отношению к подпрограмме), в которой размещено значение параметра. То есть в качестве локальной переменной выступает ссылка на глобальное размещение параметра, обеспечивающая доступ к самому значению.

Формальные параметры-переменные служат для модификации внутри подпрограммы значений соответствующих фактических параметров. Формальный параметр-переменная представляет фактическую переменную во время выполнения процедуры или функции, поэтому все изменения значения формального параметра отражаются на фактическом параметре.

Внутри подпрограммы любое упоминание формального параметра-переменной обеспечивает доступ к самому фактическому параметру. Тип фактического параметра должен быть тождественен типу формального параметра-переменной (это ограничение можно обойти через использование параметров-переменных без типа). Файловые типы могут передаваться только как параметры-переменные.

Рассмотрим пример объявления процедуры с параметром-переменной в языке Pascal (параметр, вызываемый по наименованию, в данном случае идентифицируется ключевым словом `var`):

```
// Процедура замены запятых на точки с запятой в строке
procedure NmbComma(var S: string );
var i, L:integer;
begin
   L := Length(S);
   for i := 1 to L do
     if S[i] = ',' then S[i] = ';';
   end;
```

В результате применения этой процедуры к строке S_Sentence

```
NmbComma(S_Sentence);
```

будет изменено содержимое строки. Если до вызова процедуры в строке находилось значение 'лимон, апельсин, банан', то после вызова процедуры значение S_Sentence будет 'лимон; апельсин; банан'.

Объявление параметров-переменных может (в некоторых языках) не сопровождаться указанием типа. Когда формальный параметр является параметром-переменной без типа, соответствующий фактический параметр может быть переменной любого типа, а ответственность за правильность использования параметра ложится при этом на программиста.

Внутри процедуры или функции такой параметр-переменная не имеет типа, т. е. он не совместим с переменными всех других типов до тех пор, пока ему не присвоен определенный тип.

Рассмотрим пример передачи параметров-переменных без типа (язык Pascal):

```
function VarEqual(var Source, Dest; EqSize: word): boolean;
   // Функция сравнения переменных Source и Dest длины EqSize
type
 ArrBytes = array [0 .. 1000] of byte;
   // Объявление вспомогательного локального типа данных
var
 N: integer;
begin
 N := 0;
 while (N < EqSize) and (ArrBytes(Dest)[N] = ArrBytes(Source)[N])
          do Inc(N);
 VarEqual := N = EqSize;
end;
```

Эту функцию можно использовать для сравнения любых двух переменных, размер которых не превышает 1000 байт. Например, в программе присутствуют следующие объявления:

```
type
   TVector = array[1 .. 10] of integer;
   TPoint = record
            X, Y: integer;
    end;
var
   Vector1, Vector2: TVector;
   Point1, Point2: TPoint;
```

Тогда вызов функции

```
VarEqual(Vector1, Vector2, SizeOf(Vector))
```

обеспечит сравнение массивов `Vector1` и `Vector2`.

Вызов функции

```
VarEqual(Vector1, Vector2, SizeOf(integer)*10)
```

обеспечит сравнение первых 10 элементов массивов `Vector1` и `Vector2`.

Вызов функции

```
VarEqual(Point1, Point2, SizeOf(TPoint))
```

обеспечит сравнение переменных `Point1` и `Point2`.

Вызов функции

```
VarEqual(Vector1[1], Point2.Y, SizeOf(integer))
```

обеспечит сравнение значений `Vector1[1]` и `Point2.Y`.

Пограничный характер между двумя категориями параметров носят формальные параметры-константы. С одной стороны, это параметры, которые передаются в подпрограмму по наименованию, т. е. ссылкой на глобальное размещение фактического параметра, но, с другой стороны, внутри подпрограммы действует запрет на изменение значения параметра. Использовать параметры-константы удобно вместо параметров-значений, когда параметр характеризуется большим размером занимаемой памяти.

В рассмотренном ранее примере объявления функции с параметром-значением более эффективным будет использование параметра-константы, который обеспечит передачу в процедуру адреса размещения строки:

```
// Функция вычисления количества запятых в строке
function NmbComma(const S: string ): integer;
var i, L, Nmb :integer;
begin
   Nmb := 0;
   L := Length(S);
   for i := 1 to L do
     if S[i] = ',' then inc(Nmb);
   NmbComma := Nmb;
end;
```

Аналогично объявлению параметров-переменных в объявлении параметров-констант может отсутствовать указание типа. В этом случае фактический параметр может быть переменной любого типа, и использование его внутри подпрограммы предполагает предварительное преобразование к конкретному типу.

1.5.3. Перегрузка подпрограмм

В некоторых языках программирования реализован аппарат перегрузки процедур и функций. Это означает, что можно объявить несколько процедур или функций с одинаковыми именами, но с различающимися по числу и типу параметрами. Если в соответствии с синтаксисом ЯП объявить эти процедуры (или функции) перегружаемыми, то при вызове такой процедуры или функции компилятор проанализирует передаваемые параметры, их число и тип и вызовет ту процедуру (функцию), которая соответствует данным параметрам. Например, используя синтаксис ЯП Pascal, можно объявить следующие две функции деления с одинаковым именем:

```
function Divide(X, Y: integer): integer; overload;
begin
   Result := X div Y;
end;

function Divide(X, Y: real): real; overload;
begin
   Result := X/Y;
end;
```

В первом случае параметры процедуры целочисленные, а результат (тоже целое число) получается с помощью операции целочисленного деления. Вторая функция в качестве параметров принимает вещественные числа и результатом вызова функции является, соответственно, вещественное число.

Ключевое слово `overload` в языке Pascal означает, что функции объявлены как перегружаемые. Следовательно, при вызове функции `Divide(10, 4)` будет работать первая функция, и будет получен результат, равный 2. Если же будет использован оператор `Divide(10.0, 4.0)`, вызовется вторая функция, и результат будет равен 2.5.

1.6. Рекурсивные определения и алгоритмы

Рекурсивный объект — объект, частично состоящий или определяемый с помощью самого себя. Приведем несколько общеизвестных примеров рекурсивных определений.

Определение натурального числа:

1) 0 есть натуральное число;

2) целое число, следующее за натуральным, есть натуральное число.

Функция «факториал» — *N*! (для неотрицательных целых чисел):

1) `0! = 1;`

2) `N > 0: N! = N *(N - 1)!`

Определение синтаксических конструкций с помощью формальной грамматики Бэкуса:

```
<Идентификатор> ::= <Буква> | <Идентификатор><Буква> |
                    <Идентификатор><Цифра>
```

Таким образом, рекурсивное определение позволяет с помощью конечного высказывания определить бесконечное множество объектов.

1.6.1. Рекурсивные алгоритмы

Опираясь на понятие рекурсии, можно описать бесконечное вычисление с помощью конечной рекурсивной программы, причем программа не будет содержать явных повторений. В общем виде рекурсивная программа `R` представляет собой некоторую композицию `C` из множества операторов `S` (не содержащих `R`) и самой `R`:

```
R = C[S,R]
```

Таким образом, для записи рекурсивных программ в алгоритмическом языке необходимо и достаточно иметь понятие процедуры (функции) или подпрограммы, поскольку такие конструкции позволяют создать и поименовать рекурсивный оператор и впоследствии к нему обращаться.

Если некоторая процедура (функция) `R` содержит явную ссылку па саму себя, то ее называют *прямо рекурсивной*.

Если `R` ссылается на другую процедуру (функцию) `Q`, содержащую (прямую и косвенную) ссылку на `R`, то `R` называют

косвенно рекурсивной. Косвенная рекурсивность не всегда явно определима по тексту процедуры.

Как правило, с процедурой связано множество локальных объектов (типы данных, переменные, константы, процедуры, функции), которые определены только в этой процедуре и вне ее не существуют или не имеют смысла. При каждой рекурсивной активации такой процедуры порождается новое множество локальных объектов. Они имеют те же самые имена, что и соответствующие элементы локального множества предыдущего «поколения» этой процедуры, но их значения отличны от последних. Конфликты по именам при этом разрешаются с помощью основного правила, определяющего область действия идентификаторов: *идентификатор всегда относится к самому последнему порожденному множеству переменных*. Это же правило справедливо и для параметров процедуры, по определению связанных с самой процедурой. Отложенные вызовы процедуры в процессе рекурсивного вычисления образуют стек — они размещаются один над другим, а используются в обратной последовательности.

Подобно операторам цикла, рекурсивные процедуры могут приводить к не заканчивающимся вычислениям («зацикливаниям»). Чтобы этого не случалось, рекурсивное обращение к R должно управляться условием B, которое в какой-то момент становится ложным. Поэтому более точно схему рекурсивных алгоритмов можно представить, например, в следующей форме:

```
R = if B then C[S,R] end
```

Основной способ доказательства конечности повторяющегося (циклического) процесса следующий [10].

1. На множестве переменных $x_1, x_2, ..., x_n$ задается целочисленная функция $f(x_1, x_2, ..., x_n)$, такая, что из истинности условия $f(x_1, x_2, ..., x_n) \le 0$ следует истинность условия окончания цикла (с предусловием или постусловием).

2. Доказывается, что при каждом прохождении цикла значение $f(x_1, x_2, ..., x_n)$ уменьшается.

Аналогично можно доказать и конечность рекурсии: вторым шагом необходимо доказать, что рекурсивная процедура уменьшает значение $f(x_1, x_2, ..., x_n)$, тогда отрицательное значение функции приведет к истинности условия `not B`. Например, если

- в качестве функции использовать целочисленный параметр n;

- к рекурсивной процедуре R обращаться с параметром $n - 1$;
- логическое условие B сформулировать как $n > 0$,

то окончание рекурсии будет гарантировано. Это можно выразить следующей схемой:

```
R(n) = if n > 0 then C[S,R(n - 1)] end
```

В практических приложениях важно, чтобы максимальная глубина рекурсий была не только конечна, но и достаточно мала. Так как каждая рекурсивная активация процедуры R требует памяти для размещения локальных объектов и текущих «состояние вычислений» (для обеспечения возврата по окончании новой активации R), слишком большая глубина рекурсии может приводить к переполнению стека и перераспределению памяти, что отрицательно сказывается на быстродействии выполнения программы.

Рекурсивные алгоритмы особенно подходят для задач, где обрабатываемые данные определяются в терминах рекурсии. Однако это не означает, что такое рекурсивное определение данных гарантирует бесспорность употребления для решения задачи рекурсивного алгоритма. Фактически объяснение концепций рекурсивных алгоритмов на неподходящих для этого примерах и вызвало широко распространенное предубеждение против использования рекурсий в программировании; их даже сделали синонимом неэффективности.

В программировании термин *рекурсия* часто противопоставляется *итерации*. Рекурсия и итерация являются двумя основными способами определения циклических процессов, поэтому важно выделять ситуации, где вычисляемые значения определяются с помощью простых рекуррентных отношений и в которых следует избегать алгоритмической рекурсии. Рассмотрим пример вычисления факториала:

$$f_i = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (i-1)i = (i-1)! \cdot i.$$

Естественное рекурсивное определение функции на языке программирования Pascal:

```
function factorial (n:word):word;
begin
   if n = 0 then factorial := 1
   else factorial := n * factorial(n-1)
end
```

Тип данных word обеспечивает неотрицательность целочисленного аргумента функции и результата, таким образом, factorial является функцией целочисленной неотрицательной переменной; n является формальным параметром. При выполнении функции значение аргумента n сперва сравнивается с 0. Если n > 0, то значение аргумента n должно быть умножено на значение той же самой функции factorial для значения аргумента n-1. Это означает, что вычисление первоначального функционального вызова должно быть приостановлено, и значение функции factorial(n-1) должно быть вычислено посредством той же программы. Текущее значение n и factorial (хотя последнее и не будет использоваться на самом деле) должны быть сохранены в памяти для продолжения вычислений factorial(n) после того, как будет вычислено factorial(n-1). Эта процедура откладывания будет повторена n раз, пока текущее значение n не станет равным 0. Затем отложенные вызовы возобновляются и выполняются в обратном порядке до получения результата.

Рассмотрим теперь итеративное определение той же функции:

```
function factorial (n: word): word;
var f: word;
begin
   f := 1;
   while n > 1 do
   begin f := n * f;
         n := n - 1
   end;
   factorial := f
end
```

В такой реализации функции введена еще одна неотрицательная целочисленная переменная f, которая используется при вычислении. Сначала переменной f присваивается значение 1. Далее реализован простой цикл с двумя переменными, изменяющими свое значение в ходе вычислений: до тех пор, пока n превосходит значение 1, в теле цикла вычисляются новые значения переменных f и n, а управление возвращается на проверку условия.

Таким образом, в рекурсивном алгоритме перед тем, как числа действительно начинают перемножаться, следует запомнить $2n$ значений. К тому же следует организовать правильное возобновление отложенных вызовов функции. Итеративный же

цикл потребует только вычисления $2n$ значений. Очевидно, что рекурсивное вычисление более сложно и требует больше времени и компьютерной памяти, чем итеративное.

Конечно, существуют и более сложные схемы рекурсий, которые можно и необходимо переводить в итеративную форму. Возьмем в качестве примера вычисление чисел Фибоначчи, определяемых рекурсивным соотношением:

$$fib_1 = 1,\ fib_0 = 0.$$

$$fib_{n+1} = fib_n + fib_{n-1},\ \text{для}\ n > 0.$$

Прямое переписывание соотношения приводит к рекурсивной программе:

```
function Fibonachi (n: word): word;
begin
if N = 0 then Fibonachi := 0
else if N = 1 then Fibonachi := 1
       else Fibonachi:= Fibonachi(N-1) + Fibonachi(N-2)
end;
```

Вычисление fib_n путем обращения к `Fibonachi(n)` приводит к рекурсивным активациям этой процедуры-функции. Причем, каждое обращение с $n > 1$ приводит еще к двум обращениям, т. е. общее число вызовов — число узлов дерева рекурсии — растет экспоненциально (рис. 1.19). Очевидно, что такая программа практического интереса не представляет.

Числа Фибоначчи можно вычислять и по итеративной схеме, где с помощью вспомогательных переменных $F = fib_i$

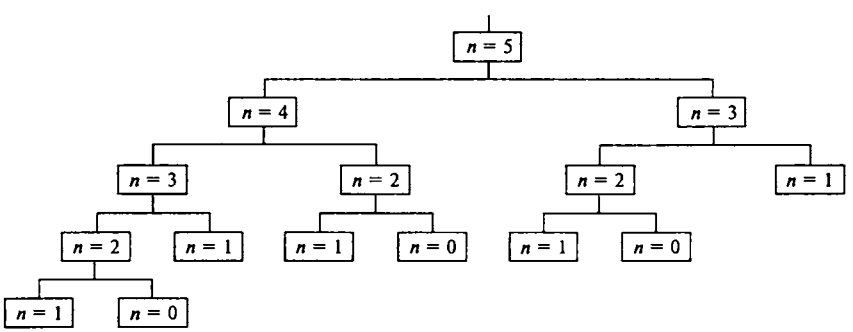


Рис. 1.19. 15 обращений к `Fibonachi(n)` при $n = 5$.

и $P = fib_{i-1}$ удается избежать повторных вычислений одной и той же величины:

```
function Fibonachi (n: word): word;
var
  F, P, R, I: word;
begin
if n = 0 then Fibonachi := 0
else if n = 1 then Fibonachi := 1
 else begin
   I := 1;
   F := 1; P := 0;
   while I < N do
     begin R := F;{Сохранение значения очередного члена ряда}
           F := F + P; {Вычисление следующего значения}
           P := R; {Восстановление предыдущего значения}
           I := I + 1
           end;
     Fibonachi := F
     end;
end.
```

Таким образом, рекомендуется избегать рекурсий там, где есть очевидное итерационное решение. Однако это не означает, что от рекурсий следует избавляться любой ценой. Существует много хороших примеров применения рекурсии. Рассмотрим алгоритмы, ищущие решение не по заданным правилам вычислений, а путем проб и ошибок — задачи так называемого «искусственного интеллекта». Обычно процесс проб и ошибок разделяется на отдельные подзадачи и естественно выражается в терминах рекурсии, требующей исследования конечного числа подзадач.

Задача о ходе коня

Для демонстрации основных методов разбиения на подзадачи и применения при этом рекурсии рассмотрим известный пример — задачу о ходе коня [10].

Дана доска размером $n \times n$, т. е. содержащая n^2 полей. Вначале на поле с координатами x_0, y_0 помещается конь — фигура, перемещающаяся по обычным шахматным правилам. Задача заключается в поиске последовательности ходов (если она существует), при которой конь точно один раз побывает на всех полях доски (обойдет доску), т. е. нужно вычислить и представить последовательность из $n^2 - 1$ ходов.

На каждой клетке доски у коня либо существует возможность сделать очередной ход, либо нет. Если возможностей сделать очередной ход несколько, необходимо проверить каждую из них. Некоторые впоследствии могут привести к тупиковым ситуациям (т. е. к отсутствию возможного хода).

Очевидный прием упростить задачу обхода n^2 полей — решать более простую: либо выполнить очередной ход, либо доказать, что никакой ход невозможен.

Используя синтаксис языка Pascal, начнем с определения алгоритма выполнения очередного хода. Первый его вариант может быть таким:

```
procedure TryNextMove;
begin <инициализация выбора хода>;
repeat  <выбор очередного кандидата из списка ходов>;
if <подходит> then <запись хода>;
if <доска не заполнена> then TryNextMove;
if <неудача> then <уничтожение предыдущего хода>
until (<был удачный ход>) or (<кандидатов больше нет>)
end TryNextMov;
```

Чтобы описать этот алгоритм более детально, необходимо выбрать типы и сформировать структуры данных.

Шахматная доска аналогична двумерному массиву. Для индексирующих значений подходит тип `integer`. Содержимое полей доски будем использовать для хранения истории передвижения по доске, т. е. в каждом отдельном элементе двумерного массива будем хранить номер хода, который привел коня на соответствующее поле доски. Тогда массив для хранения данных о передвижении коня может быть описан следующим образом:

```
var h: array [1 .. n, 1 .. n] of integer;
```

Очевидно, могут быть предложены следующие соглашения о содержимом отдельного элемента массива:

`h[x,y] = 0` — поле с координатами `(x,y)` еще не посещалось;

`h[x,y] = i` — поле с координатами `(x,y)` посещалось на `i`-м ходу.

Теперь нужно выбрать параметры, определяющие начальные условия следующего хода и результат (если ход сделан). В первом случае достаточно задавать координаты поля `(x,y)`, откуда следует ход, и число `i`, указывающее номер хода (для фиксации в массиве `h`). Для результата требуется параметр логического

типа, который принимает значение «истина», если ход был возможен.

Далее на основе принятых решений уточним некоторые операторы:

- условие «доска не заполнена» можно переписать как $i < n^2$;
- если ввести две локальные переменные u и v для возможного хода, определяемого в соответствии с правилами «прыжка» коня, то условие «возможен» можно представить как логическую конъюнкцию условий «новое поле находится в пределах доски» и «новое поле еще не посещалось»:

```
(1 <= u) and (u <= n) and (1 <= v) and (v <= n) and
(h[u,v] = 0);
```

- фиксация допустимого хода выполняется с помощью оператора присваивания

```
h[u,v] := i;
```

- отмена хода:

```
h[u,v] := 0;
```

- если ввести локальную переменную `ql` и использовать ее в качестве параметра-результата при рекурсивных обращениях к этому алгоритму, то `ql` можно подставить вместо «есть ход».

После сделанных уточнений получаем следующий вариант:

```
procedure TryNextMove (i: integer; x, y: integer;
                       var y: boolean);
var u, v: integer;
    ql: boolean;
begin <инициация выбора хода>;
repeat <(u,v) — координаты следующего хода, определяемого
            правилами шахмат>;
if (1 <= u) and (u <= n) and (1 <= v) and (v <= n) and
   (h[u,v] = 0)
then h[u,v]: = i;
     if i < n*n then TryNextMove(i+1, u, v, ql);
        if not ql then h[u,v]: = 0 else ql: = true
until ql or <других ходов нет>;
q:=ql
end < TryNextMove>.
```

До этого момента программа создавалась совершенно независимо от правил, управляющих движением коня. На самом

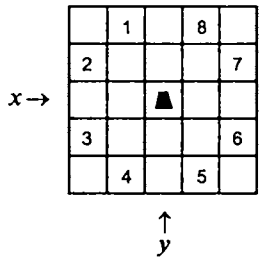


Рис. 1.20. Восемь возможных ходов коня

деле, если задана начальная пара координат x, y, то для следующего хода u, v существует восемь возможных кандидатов. На рис. 1.20 они пронумерованы от 1 до 8. Получать u и v из x и y можно, прибавляя к последним разности между координатами, хранящиеся в одном (для обеих координат) либо в двух (отдельно для каждой координаты) массивах разностей. Выберем способ раздельного хранения и обозначим массивы через dx и dy. Будем считать, что они соответствующим образом инициированы:

```
dx[1]:= 2;    dx[2]:= 1;    dx[3]:= -1;    dx[4]:= -2;
dx[5]:= -2;   dx[6]:= -1;   dx[7]:= 1;     dx[8]:= 2;
dy[1]:= 1;    dy[2]:= 2;    dy[3]:= 2;     dy[4]:= 1;
dy[5]:= -1;   dy[6]:= -2;   dy[7]:= -2;    dy[8]:= -1;
```

Для нумерации очередного хода-кандидата можно использовать индекс k. Первый раз параметрами обращения к рекурсивной процедуре являются значения x0 и y0 — координаты поля, с которого начинается обход. Этому полю должно быть присвоено значение 1, остальные поля помечаются как свободные (т. е. значением 0):

```
for i := 1 to n do for j := 1 to n do h[i,j] := 0;
...
h[x0,y0] := 1;
```

Состав глобальных переменных программы:

```
var x0, y0, n : integer;
    Nsqr: integer;      // переменная для хранения значения n²
    q: boolean;
    dx, dy: array [1..8] of integer;
    h: array [1..8,1..8] of integer;
```

Окончательный вариант рекурсивной процедуры обхода доски следующий:

```
procedure TryNextMove(i,x,y: integer; var q: boolean);
var k, u, v: integer;
  _ ql : boolean;
begin k := 0;
repeat  k := k+1;
   ql := false;
   u := x + dx[k]; v:= y + dy[k];
   if (1<=u) and (u<=n) and (1<=v) and (v<=n) and (h[u,v] = 0)
    then begin h[u,v] := i;
               if i < Nsqr
               then begin  TryNextMove(i+1, U, V, ql);
                           if not ql then h[u,v] := 0 end
               else ql := true end
until (ql) or (k = 8);
q:=ql
end;
```

Программа, решающая задачу обхода поля ходом коня, окончательно состоит из следующих шагов:

1) инициализация глобальных переменных;
2) ввод значений x0, y0, n;
3) вызов процедуры TryNextMove;
4) вывод полученного результата.

На рис. 1.21 приводятся решения для исходных позиций: (1, 3) при n = 5 и (2, 4) при n = 6.

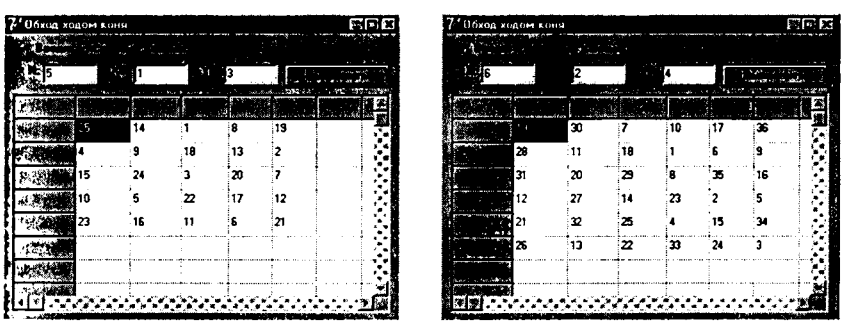


Рис. 1.21. Два возможных обхода доски

Итак, характерное свойство таких алгоритмов заключается в следующем:

- формируется последовательность шагов в направлении общего решения;

- все шаги фиксируются (записываются) таким образом, чтобы позже можно было вернуться, отбрасывая шаги, которые не ведут к общему решению (заводят в тупик).

Такой процесс называется *возвратом* или *откатом* (backtracking). Если предположить, что число потенциальных шагов конечно, то из схемы процедуры `TryNextMove` можно вывести универсальную схему:

```
procedure Try;
begin <инициация выбора кандидата>;
repeat <выбор очередного кандидата>;
if <подходит> then <запись кандидата>;
if <решение неполное> then try;
if <неудача> then <стирание записи>;
until <удача> or <кандидатов больше нет>
end Try
```

1.6.2. Рекурсивный тип данных

Ранее (п. 1.3—1.4) были рассмотрены основные структурированные типы данных и наиболее распространенные структуры данных. Используя суперпозицию типов данных, можно не только получить сложные структуры данных, но и осуществить рекурсивное определение типа.

Рассмотрим, например, бинарное дерево, конечные вершины (листья) которого имеют тип `integer`. Такая структура данных может быть описана следующим образом:

```
type tree = union
            nterm : record
                      left, right : tree
            end;
            term : integer
            end;
```

Для описания типа данных `tree` применяется в свою очередь тип `tree`. Наиболее естественным способом толкования такого описания является рекурсивное определение `tree`. Множество значений такого типа данных есть множество бинарных деревьев. Запишем последовательно несколько данных, принадлежащих этому множеству (рис. 1.22).

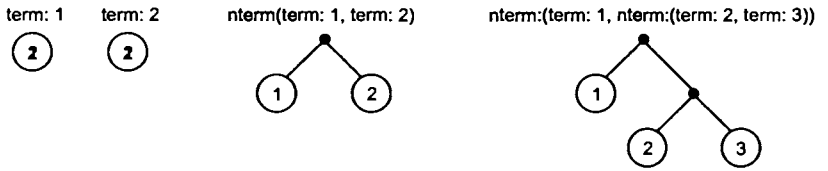


Рис. 1.22. Рекурсивный тип данных

Здесь необходимо обратить внимание на то, что `term` и `nterm` являются признаками, указывающими, является ли узел листом дерева.

Обработка данных рекурсивного типа обычно наиболее целесообразна, когда она соответствует схеме определенного рекурсивного типа и осуществляется с помощью рекурсивных функций и процедур.

Рассмотрим определенное выше бинарное дерево. Если заданы данные `d` типа `tree`, то можно определить рекурсивную функцию `sum`, которая суммирует значения вершин-листьев `d`:

```
function sum (d: tree): integer;
begin
     with d do
          nterm: s: = sum(d.left) + sum(d.right);
          term: s:= d
     end
end
```

Необходимо обратить внимание на то, что вид такой функции сходен с видом описания типа данных `tree`.

1.7. Структура программы на языке высокого уровня

Исходная программа, как правило, состоит из следующих частей (впервые эти требования были сформулированы в языке Cobol):

- раздела идентификации — области, содержащей наименование программы, а также дополнительную информацию для программистов и/или пользователей;
- раздела связи — фрагмента текста, описывающего внешние переменные, передаваемые вызывающей программой (если таковая имеется), т. е. ту часть исходных данных, которая

обязательно поступает на вход программы при ее запуске. Эти переменные часто называют параметрами программы;

- раздела оборудования (среда) — описания типа ЭВМ, процессора, требований к оперативной и внешней памяти, существенных с точки зрения выполнимости программы[2];
- раздела данных — идентификации (декларация, объявление, описание) переменных, используемых в программе, и их типов. Понятие типа позволяет осуществлять проверку данных на совместимость в операциях еще на этапе трансляции программы и отвергнуть недопустимые преобразования;
- раздела процедур — собственно программной части, содержащей описание процессов обработки данных. Элементами процедуры являются операторы и стандартные функции, входящие в состав соответствующего языка программирования.

Необходимо отметить, что конкретные ЯП могут не требовать наличия всех вышеперечисленных разделов исходного модуля. В некоторых случаях описания переменных могут размещаться произвольно в тексте или опускаться, при этом тип переменной определяется компилятором, исходя из системы умолчаний; есть средства программирования, в которых тип переменной задается в момент присвоения ей значения другой переменной или константы и т. д. Существуют фрагменты описания данных, которые могут быть отнесены как к разделу данных, так и к разделу оборудования (указания на устройство, длину и формат записи, организацию файла и т. п.). Так, например, программа на языке Pascal представляет собой последовательность заголовка, раздела объявлений и тела программы. Структурная схема программы на языке Pascal изображена на рис. 1.23. Следует отметить, что в современных реализациях языка (Turbo Pascal, Object Pascal) заголовок программы необязателен и не обрабатывается компилятором; порядок размещения разделов объявлений произвольный (в отличие от стандарта, где этот порядок строго фиксирован); в программе может быть

[2] Дело в том, что даже среди семейства однотипных ЭВМ могут существовать отличия в наборе машинных инструкций (команд), средств программирования ввода-вывода, кодированного представления данных, поэтому описание среды, приводимое в данном разделе, оказывается необходимым транслятору с языка с точки зрения оптимизации выполнения рабочей программы или оценки возможности ее создания.

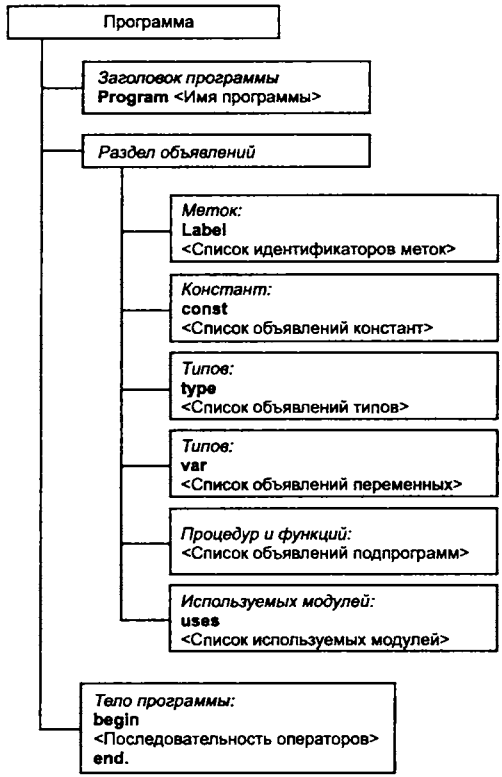


Рис. 1.23. Структурная схема программы на языке Pascal

указано несколько одинаковых разделов объявлений и существует раздел объявления используемых модулей (в стандарте языка этот раздел отсутствовал).

1.7.1. Область видимости и время жизни программных элементов

Такие элементы программы на ЯВУ, как переменные, константы, процедуры и функции, могут использоваться только в области их видимости. Область видимости определяется положением объявления элемента внутри программы. Правила использования переменных, констант, процедур и функций в программе опираются на понятие «блок».

Блок представляет собой последовательность объявлений, определений и операторов, которая заключена в так называемые операторные скобки (в соответствии с синтаксисом конкретного языка). Обычно выделяют два типа блоков: составной оператор и определение процедуры или функции. Любой блок может содержать в себе другие (вложенные) блоки.

Время жизни — это период выполнения программы, в котором существует переменная, константа, процедура или функция.

Область видимости конкретного программного элемента относит его к классу локальных (с локальной областью видимости) или глобальных (с глобальной областью видимости) элементов. Элемент с локальной областью видимости хранит и определяет значение только в блоке, в котором он был определен или объявлен. Локальному элементу всегда выделяется новое место в памяти каждый раз, когда программа входит в блок, и хранимая величина теряется, когда программа выходит из блока. Если область видимости элемента глобальна, он хранит и определяет значение на всем протяжении выполнения программы.

Следующие правила определяют время жизни программного элемента:

- элементы, которые объявлены на внутреннем уровне (в блоке), обычно имеют локальное время жизни;
- элементы, которые объявлены на внешнем уровне (вне всех блоков программы), всегда имеют глобальное время жизни.

Область видимости элемента определяет ту часть программы, в которой на него можно сослаться по имени. Элемент доступен только в своей области видимости, которая может быть ограничена (в порядке усиления ограничений) файлом (или модулем), процедурой или функцией, блоком.

Объявленный на внешнем уровне элемент имеет областью видимости весь файл и доступен из любого места файла. Если его объявление происходит в блоке (включая список формальных параметров в определении процедуры или функции), область видимости элемента ограничивается данным блоком и вложенными в него блоками. Имена формальных параметров в списке параметров процедуры или функции обычно имеют область видимости только от объявления параметра до конца объявления функции.

Про элемент говорят, что он *глобально доступен*, если он доступен из всех мест программы.

Следующие правила определяют доступ к элементам в программе:

1. Элементы, которые были объявлены или определены на глобальном уровне (т. е. вне всех блоков программы), доступны от точки их объявления или определения до конца исходного файла. Можно использовать соответствующие объявления для того, чтобы эти переменные стали доступны и из других исходных файлов.

2. В общем случае элементы, которые объявлены или определены на локальном уровне (т. е. внутри блока), доступны всем вложенным блокам. Однако в этом случае говорят о «вложенности» доступа. Например, блок, который вложен в другой блок, может содержать объявления переменных, идентификаторы (имена) которых могут совпадать с переменными внешнего блока. Такие переопределения доминируют только во внутреннем блоке. После выхода из внутреннего блока будет восстановлено определение внешнего блока.

В табл. 1.8 суммируются основные факторы, определяющие время жизни и область видимости.

Таблица 1.8. **Краткий обзор времени жизни и области видимости**

Место объявления элемента	Время жизни	Область видимости
Раздел объявлений головного файла программы	Глобальное	Вся программы
Раздел объявлений отдельного модуля (файла) программы	Глобальное	От точки объявления до конца модуля (файла), а также во всех модулях, имеющих объявленный доступ к текущему
Раздел объявлений блока	Локальное	От точки объявления до конца блока (включая все вложенные блоки)

Как видно из табл. 1.8, элементы, объявленные в блоке, не только видны именно в нем, но и локальны, т. е. существуют только в нем. Они создаются (для них отводится область памяти) только при передаче управления в данный блок и уничтожаются, когда управление покидает этот блок.

Если во вложенном блоке объявлен элемент, имя (идентификатор) которого совпадает с именем элемента внешнего блока, то во вложенном блоке внешний элемент не виден, а виден только элемент, объявленный внутри блока. Например, если некоторая переменная с идентификатором `I` объявлена сначала во внешнем блоке, а потом во внутреннем, то значение, которое

переменная получила во внешнем блоке (и сама переменная) недоступно во внутреннем блоке. Переменная внутреннего блока создается и действует только в этом блоке и теряет свое значение при выходе из него.

1.7.2. Модули

Основа для массового промышленного программирования была создана с разработкой методов построения программ. Одной из первых и наиболее широко применяемых технологий программирования стало *структурное программирование*. Этот метод до сих пор не потерял своего значения для определенного класса задач.

Структурное программирование

Структурный подход базируется на двух основополагающих принципах:

- использование процедурного стиля программирования;
- последовательная декомпозиция алгоритма решения задачи сверху вниз.

В соответствии с этим подходом задача решается путем выполнения следующей последовательности действий. Первоначально задача формулируется в терминах ввода данных — вывода результата: на вход программы подаются некоторые данные, программа работает и выдает ответ (рис. 1.24). После этого начи-

Рис. 1.24. Верхний уровень структурного подхода

нается последовательное разложение всей задачи на отдельные более простые действия. При этом на любом этапе декомпозиции программу можно проверить, применяя механизм так называемых «заглушек» — процедур, имитирующих вход и/или выход процедур нижнего уровня. «Заглушки» позволяют проверить логику верхнего уровня до реализации следующего, т. е. на каждом шаге разработки программы можно иметь работающий каркас, который постепенно обрастает деталями.

Программный модуль

Структурное программирование в свою очередь определило значение модульного построения программ (т. е. разбиения монолитных программ на группу отдельных модулей) при разработке больших проектов. Модульное программирование возникло еще в начале 1960-х годов. *Модуль* представлял собой последовательность логически связанных фрагментов, оформленных как отдельная часть программы, а само модульное программирование характеризовалось следующими преимуществами [32]:

1. Большую программу могли писать одновременно несколько исполнителей.
2. Стало возможным создавать библиотеки наиболее употребительных подпрограмм.
3. Стала проще процедура загрузки в оперативную память большой программы, требующей сегментации.
4. В программе, состоящей из модулей, много естественных контрольных точек для наблюдения за выполнением проекта.
5. Стало возможным более полное тестирование.
6. Упростился процесс проектирования и последующего изменения программы.

Последний пункт был особенно важен в связи с тем, что стоимость сопровождения и модификации программ составляет значительную часть общих расходов на обработку данных.

Наряду с этими преимуществами были выявлены и недостатки, которые могут приводить к возрастанию стоимости программы:

1) может увеличиться время исполнения программы;
2) может возрасти размер требуемой памяти;
3) может увеличиться время компиляции и загрузки;
4) проблемы организации межмодульного взаимодействия могут оказаться довольно сложными.

Приведем основные свойства модуля, которые рекомендовалось соблюдать разработчикам:

1. Модуль является частью результата *совместной компиляции*. Он может активизироваться операционной системой или быть подпрограммой, вызываемой другим модулем.

2. На внутренность модуля можно ссылаться с помощью имени, которое называется именем модуля.

3. Модуль должен возвращать управление тому, кто его *вызвал*.

4. Модуль может обращаться к другим модулям.

5. Модуль должен иметь один *вход* и один *выход*.

6. Модуль не должен сохранять историю своих вызовов для управления своим функционированием.

7. Модуль реализует единственную функцию: вполне определенное преобразование исходных данных в результат, осуществляемое в процессе исполнения данного модуля. Его функция может быть выражена одной фразой, например:

«Редактировать запрос»;

«Загрузить файл»;

«Обратить матрицу»;

«Напечатать отчет».

Таким образом, проектирование программы предполагает сначала функциональную декомпозицию поставленной задачи.

Последнее свойство модуля определило технологию *нисходящего проектирования* программ.

В основе нисходящего проектирования лежит процесс построения схемы иерархии, отражающей функции и *взаимодействие* модулей. Чтобы создать схему иерархии, т. е. спроектировать структуру модулей, следует начать с вершины и идти вниз (отсюда и термин «нисходящее проектирование»), чтобы один модуль в программе управлял выполнением остальных модулей.

Рассмотрим, например, диалоговую систему, которая обрабатывает запросы на поиск информации. В схеме иерархии для программы обработки запросов прежде всего могла бы появиться самая общая функция (рис. 1.25).

Затем появляются модули, необходимые для идентификации и детализации общей функции (рис. 1.26).

Здесь все модули *передают* информацию в главную управляющую программу «Обработать запрос». Эта программа активизирует подчиненные модули, проверяет их результаты, принимает решения и осуществляет управление.

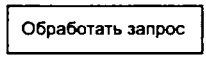


Рис. 1.25. Пример общей функции

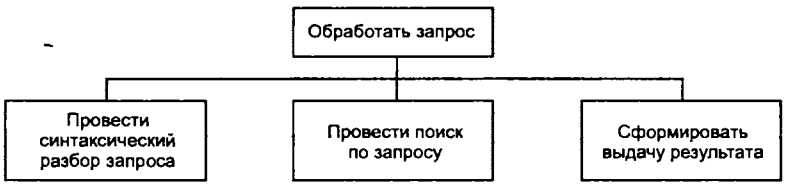


Рис. 1.26. Пример детализации общей функции

Подобный подход хорош для небольших или простых программ. Однако с возрастанием сложности программы растет и сложность главного модуля. В программе, где одна подпрограмма управляет сотней модулей, главный модуль будет настолько сложным, что его трудно будет отлаживать и изменять. Тогда в управляющей программе появляются подпрограммы второго уровня, подчиненные главной подпрограмме. Эти подпрограммы будут осуществлять некоторые функции управления и принятия решений. В этом случае структура модуля становится иерархической.

Дальнейшее развитие модульного программирования видоизменило задачи модулей: они используются для создания библиотек, которые могут включаться в различные программы (при этом становится необязательным иметь в наличии исходный код). Имя модуля по-прежнему используется при ссылке на модуль, но сам модуль чаще представляет собой набор описаний констант, типов, переменных, процедур и функций, которые являются глобальными по отношению к основной программе. Однако в языках программирования единственным способом структуризации программы по-прежнему оставалось ее составление из подпрограмм и функций.

В процессе проектирования и разработки программных комплексов тем не менее возникали следующие проблемы:

1. Развитие языков и методов программирования не успевало за все более растущими потребностями в прикладных программах. Единственным реальным способом снизить временные затраты на разработку был метод многократного использования разработанного программного обеспечения, т. е. проектирование новой программной системы на базе разработанных и отлаженных ранее модулей.

2. Ускорение разработки программного обеспечения требовало решения проблемы упрощения его сопровождения и модификации.

3. Не все задачи поддаются алгоритмическому описанию по требованиям структурного программирования, поэтому в целях упрощения процесса проектирования необходимо было решить проблему приближения структуры программы к структуре решаемой задачи.

Решение перечисленных проблем было найдено в рамках создания объектно-ориентированного подхода к программированию (подробнее см. п. 2.2 гл. 2) и породило три его основных достоинства: упрощение процесса проектирования программных систем, легкость их сопровождения и модификации и минимизирование времени разработки за счет многократного использования готовых модулей.

Компонент

Использование библиотек классов объектного программирования повышает скорость разработки программ, но, с другой стороны, требует определенных усилий для изучения этих библиотек и понимания того, как они устроены. Кроме того, библиотека классов должна быть написана на том же языке программирования, что и разрабатываемая программа. Конечно, существуют способы сопряжения разных языков программирования, но все равно, для того, чтобы использовать, например, для программы, написанной на языке Pascal, библиотеку классов C++, необходимо написать программу с вызовами нужных функций или порождением необходимых классов.

Подобные неудобства привели к появлению концепции *компонента* — программного модуля или объекта, который готов для использования в качестве составного блока программы и которым можно визуально манипулировать во время разработки программы.

Компонент — это объект, объединяющий состояние и интерфейс (способ взаимодействия), который позволяет включать компоненты в современные среды разработки приложений. При этом не важно, на каком языке программирования реализован компонент. Он должен просто удовлетворять определенным внешним параметрам и быть нейтральным по отношению к языку программирования, чтобы его можно было использовать в

программе на любом языке, поддерживающем компонентную технологию. Так, например, компоненты стандарта ActiveX могут быть одинаково успешно включены в программу, реализованную в среде Visual Basic, и в приложение, разработанное средствами Delphi.

Компоненты графического интерфейса, управляемые событиями, являются основным «строительным» материалом при разработке приложений средствами графических редакторов.

Общий принцип работы приложения с графическим интерфейсом может быть представлен схемой, изображенной на рис. 1.27. Работающее приложение находится в состоянии ожидания события, которое возникает в результате взаимодействия

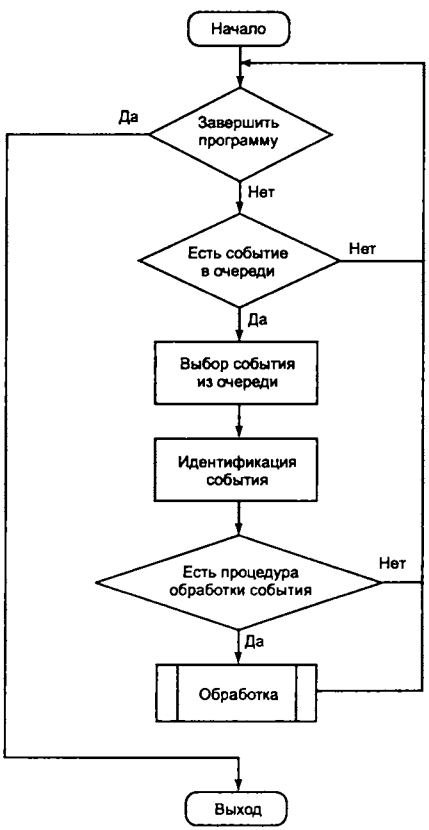


Рис. 1.27. Схема работы приложения с графическим интерфейсом

пользователя с элементами управления графического интерфейса приложения. При получении события от компонента программа передает управление процедуре обработки этого события (если таковая предусмотрена набором функций приложения).

1.8. Спецификация программ и стандартизация ЯП

Спецификация программы — это детальное (начальное или промежуточное) описание задачи, которую должна решать проектируемая программа. Если программа — это задание для вычислительной машины, написанное программистом, то спецификация — это задание для программиста, написанное системным аналитиком.

Спецификация должна быть таким описанием конкретной задачи, которое, во-первых, наиболее естественно для данной задачи, и, во-вторых, понятно как заказчику (потребителю программной продукции), так и программисту, создающему программу.

Спецификация — точное, однозначное, недвусмысленное описание, составить которое означает подобрать понятия, наиболее адекватные рассматриваемой задаче.

Различают *функциональный* и *эксплуатационный* аспекты спецификации. К функциональному аспекту, например, относятся:

- разбиение задачи на подзадачи;
- описание входных и выходных данных;
- описание объектов, участвующих в задаче;
- описание связей между объектами;
- описание процессов взаимодействия с внешней средой;
- описание реакций на исключительные ситуации и т. д.

Эксплуатационный аспект рассматривает такие вопросы, как:

- скорость работы программы;
- ресурсы, требуемые для решения задачи;
- характеристики аппаратных средств;
- специальные требования к надежности и безопасности и т. д.

Спецификации могут появляться на всех этапах так называемого «жизненного цикла» программного продукта (подробнее о понятии «жизненного цикла» см. в гл. 3). Рождается спецификация на этапе проектирования и занимает обычно промежуточное

положение между начальными требованиями и готовой программой. Дополненные и уточненные требования — это уже спецификация. Однако зачастую переход от спецификации к программе — сложный и длительный социальный процесс, в котором участвуют как заказчики, так и разработчики. Поэтому иногда полезно различать *внешнюю* и *внутреннюю* спецификации. Внешняя спецификация обращена к внешнему пользователю, потребителю программной продукции, а внутренняя — к внутреннему пользователю, разработчику.

Исходная спецификация представляет собой документ, который служит заданием на разработку программы. Хорошая спецификация при этом сокращает вероятность возникновения ошибок недопонимания разработчиками того, что хочет заказчик. На этапе тестирования программы спецификация служит для выяснения, выполняет ли программа заявленные функции. На этапе сопровождения программы спецификация облегчает процессы внесения необходимых изменений. Таким образом, спецификацию можно назвать «человекоориентированным» двойником программы, который сопровождает ее на всех этапах жизни.

1.8.1. Понятийные средства спецификации программ

Работа с понятиями является главной в построении и использовании спецификаций. Понятия, из которых и с помощью которых строятся спецификации, принято называть понятийными средствами спецификации программ [1].

Работа над выразительными возможностями спецификаций ведет к созданию новых понятийных средств. Рассмотрим далее некоторые исторически сложившиеся классы таких средств.

Табличные средства

В основе понятия «таблица» лежат идеи двумерности и упорядоченности. В этом смысле таблицу можно определить как объект, состоящий из конечного числа элементов, образующих *строки* и *столбцы*, причем положение каждого элемента в таблице определяется указанием имени (номера, индекса) строки и столбца.

Таблицы используются для описания конечных функций, конечных отношений, табличных структур данных (включая мно-

Стаж > 15 лет	Да	Да	Да	Да	Нет	Нет	Нет	Нет
Возраст > 50 лет	Да	Да	Нет	Нет	Да	Да	Нет	Нет
Пол = мужской	Да	Нет	Да	Нет	Да	Нет	Да	Нет
Премия (%)	90	80	70	60	50	0	0	0

Рис. 1.28. Таблица решений

жества записей). Например, для описания конечных функций, аргументы которых представляют собой условия, а значения — некоторые действия, используются так называемые *таблицы решений*. На рис. 1.28 представлена таблица решений, определяющая процент начисления премии в зависимости от конкретных условий.

Контекстно-свободные грамматики (КС-грамматики)

КС-грамматика — один из способов точного описания синтаксиса выражений, т. е. правил, определяющих их форму и структуру. Такие грамматики широко используются для описания синтаксиса языков программирования и других формальных языков.

Рассмотрим, например, КС-грамматику в форме Бэкуса-Наура, определяющую арифметические выражения, построенные из операций сложения (+) и умножения (*) и переменной *x*.

```
<арифметическое выражение> ::= <слагаемое> |
          <арифметическое выражение> + <слагаемое>
<слагаемое> ::= <множитель> | <слагаемое> * <множитель>
<множитель> ::= x |(<арифметическое выражение>)
```

В такой грамматике формализовано понятие приоритета операции умножения над операцией сложения.

КС-грамматика не просто задает язык описания синтаксических правил, но и приписывает выводимым цепочкам синтаксические структуры в виде помеченных деревьев, которые называются *деревьями синтаксического разбора*. Каждой цепочке в выводе соответствует свое дерево в последовательности деревьев. В грамматике, определяющей арифметическое выражение, выводу

```
<арифметическое выражение> =>
<арифметическое выражение> + <слагаемое> =>
<арифметическое выражение> + <слагаемое> * <множитель> =>
<арифметическое выражение> + <слагаемое> * x => ... => x + x * x
```

Рис. 1.29. Дерево вывода арифметического выражения

соответствует последовательность деревьев, приводящая к дереву, представленному на рис. 1.29.

Логические средства

Под логическими средствами подразумеваются прежде всего средства описания утверждений, логических связей между ними, способы построения утверждений и т. п.

Утверждения в логике первого порядка записываются в виде *формул*. Атомарная формула — это выражение вида $P(T_1, T_2, ..., T_n)$, где P — предикатный символ, а $T_1, T_2, ..., T_n$ — термы (выражения, не содержащие предикатных символов). Предикат можно рассматривать, как утверждение с переменными, истинность которого зависит от значения переменных. Например, предикат `Больше(x,y)` принимает значение «истина» при $x = 10$, $y = 2$ и «ложь» при $x = 2$, $y = 10$.

Из атомарных формул с помощью логических связок (конъюнкция, дизъюнкция, импликация, эквивалентность, отрицание) и кванторов (всеобщности, существования) строятся более сложные формулы. Например, в соответствии с таблицей решений (см. рис. 1.28) условие получения премии размером 50 % может быть записано так:

```
Стаж < 10 & Пол = 'мужской' & Возраст > 50
```

Графовые средства

Графовые средства (графы, сети, диаграммы) описывают взаимодействия объектов с относительно простыми видами связей, допускающими наглядное графическое изображение. Связи

обычно изображают линиями или стрелками, а объекты — точками, прямоугольниками, кругами, буквами и т. п.

Получаемые таким образом изображения отражают некоторые понятия и являются на самом деле формой синтаксиса понятий.

Рассмотрим наиболее распространенные графовые средства.

Граф $G = <V, R>$ можно рассматривать как бинарное отношение R на множестве V, которое определяет множество упорядоченных пар элементов из V. Особый класс бинарных отношений, которые называются *иерархическими*, представляют собой *деревья*. Структуры данных, представляющие дерево, были рассмотрены в п. 1.3.

Деревья наравне с таблицами играют важную роль в описаниях связей между элементами. Чтобы указать место элемента (узла) в древовидной структуре, достаточно указать путь к нему из корневого узла. Если узлам и дугам дерева присвоены имена, то такое дерево называется *помеченным*.

На рис. 1.30 представлены способы описания в виде деревьев типа и значения объкта, описание которого в языке Pascal следующее:

```
Record
   F1: Real;
   R: record
      F2: Boolean;
      F3: Integer;
   End
End
```

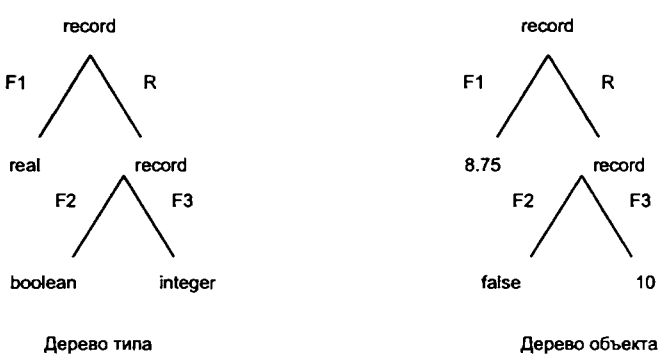


Рис. 1.30. Пример структуры составного иерархического объекта

Синтаксические диаграммы

При описании синтаксиса формальных языков используются так называемые *синтаксические диаграммы*, в графической форме определяющие последовательности построения правильных текстов. Способы графического представления синтаксических диаграмм могут быть различными. На рис. 1.31, например, приведена одна из форм описания синтаксиса арифметического выражения, представленного выше грамматикой Бэкуса-Наура. На такой синтаксической диаграмме помечены дуги, а узлы делятся на входные (с жирной точкой), терминальные (с кружочком) и промежуточные.

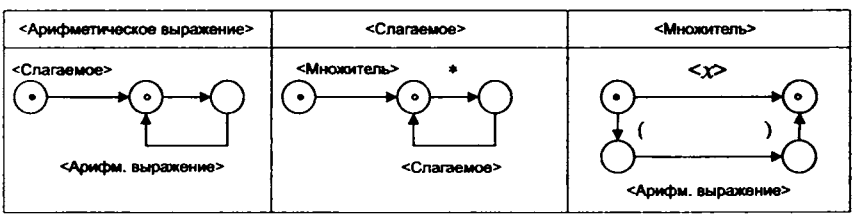


Рис 1.31. Синтаксическая диаграмма арифметического выражения

Для описания алгоритмов используется способ изображения связей между действиями с помощью *блок-схем*. Блок-схема состоит из узлов, метки которых интерпретируются как действия (операторы, блоки, подпрограммы, модули и т. п.), и дуг, которые в свою очередь тоже могут быть помечены. Если из одного узла выходит несколько дуг, то все они могут иметь разные метки. Выделяется один начальный узел и один или несколько завершающих. Основные конструкции, использующиеся при составлении блок-схем, рассмотрены в п. 1.1.

На рис. 1.32 приведена блок-схема решения задачи вычисления наибольшего общего делителя (НОД) двух натуральных чисел A и B с применением алгоритма Евклида.

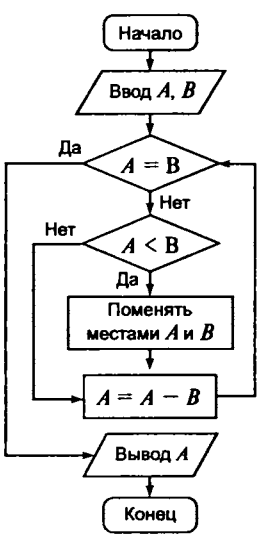


Рис. 1.32. Блок-схема алгоритма Евклида вычисления НОД

Диаграмма «объектов-связей»

Диаграмма «объектов-связей» представляет собой помеченный граф, в котором множество узлов разбито на два подмножества: объекты и связи. Узлы разных типов обычно по-разному изображаются графически, например, узлы-объекты — прямоугольниками, а узлы-связи — овалами. Объект может быть простым или составным. Из узла объекта, который является составным, выходят дуги, ведущие в узлы-объекты, интерпретируемые как элементы или атрибуты составного объекта. Дуги при этом помечаются именами атрибутов. Из узла, интерпретируемого как связь, выходят дуги, ведущие к объектам связи. Эти дуги помечаются именами, характеризующими связь.

На рис. 1.33 показана диаграмма объектов-связей, в которой в виде узлов-связей представлены отношения «Учится» и «Входит в состав», а остальные узлы («Студент», «Факультет», «Вуз», «Строка», «Целое») являются объектами. При этом узлы типа «Строка» и «Целое» характеризуют атрибуты составных объектов «Студент», «Факультет» и «Вуз».

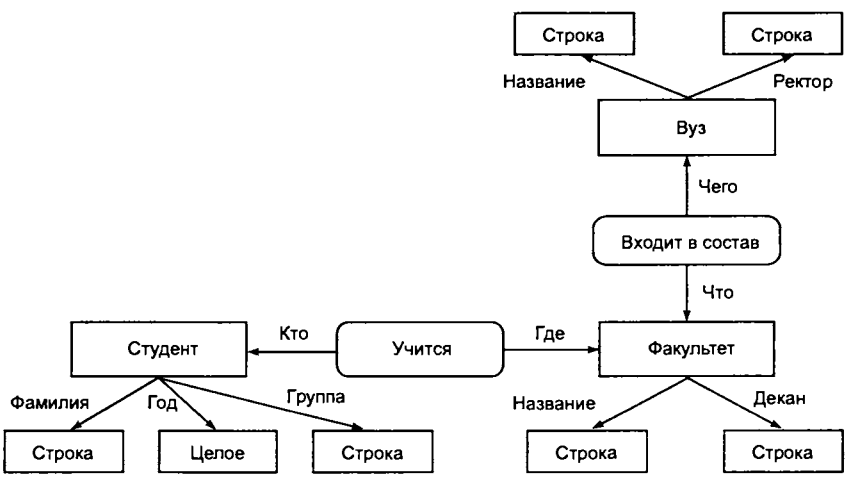


Рис. 1.33. Диаграмма объектов—связей

К диаграмме «объектов-связей» близко понятие *семантической сети*, происходящее из области искусственного интеллекта и представления семантики естественных языков. Семантическая сеть всегда предполагает некоторую семантическую надстройку (интерпретацию) над просто сетью.

Многие варианты определения понятия семантической сети объединяет стремление представить графически чуть ли не всю информацию о моделируемой ситуации. На рис. 1.34 изображен фрагмент семантической сети, выражающей примерно следующую информацию: «Николаю принадлежит автомобиль ГАЗ 3110 с номером «М 736 КО». Николай является агентом (собственником) в событии Владелец. Николай является также элементом множества Мужчины, которое в свою очередь есть подмножество множества Люди и т. д.

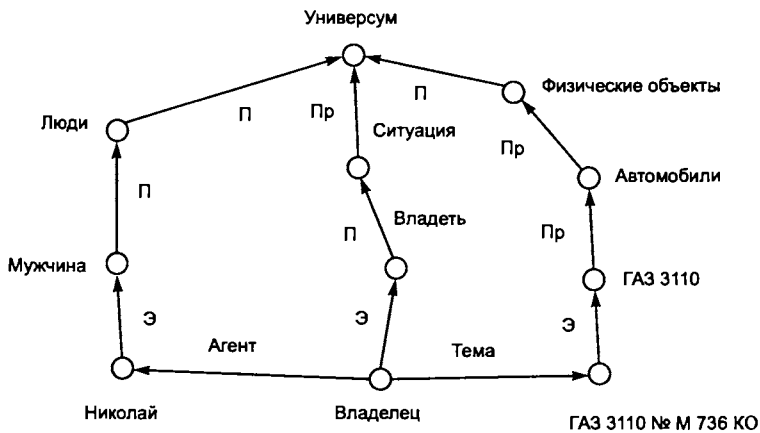


Рис. 1.34. Фрагмент семантической сети:
Э — элемент множества; П — подмножество; Пр — подмножество различное (не пересекающееся с другими)

1.8.2. Стандартизация языков программирования

Концепция языка программирования неотрывно связана с его реализацией. Для того чтобы компиляция одной и той же программы различными компиляторами всегда давала одинаковый результат, разрабатываются стандарты языков программирования. Стандартизация языков программирования создает предпосылки для повышения мобильности программного обеспечения для компьютеров любой архитектуры. Существует ряд организаций, целенаправленно занимающихся вопросами стандартизации:

- Американский национальный институт стандартов ANSI (American National Standards Institute);

- Институт инженеров по электротехнике и электронике IEEE (Institute of Electrical and Electronic Engineers);
- Организация международных стандартов ISO (International Standards Organization).

Исторически наиболее полно и тщательно стандартизировались языки программирования третьего поколения — C, Fortran, Ada и другие. В стандартах изложены:

- синтаксис ЯП;
- семантические правила интерпретации языковых конструкций;
- правила представления текстов программ;
- правила описаний данных;
- методы аттестации процессоров ЯП.

Языки программирования третьего поколения обеспечивают разработчикам возможность описания программ в терминах функций, процедур и данных. Основные различия между ЯП сосредоточены в методах и средствах описания структуры и типов используемых данных. Имеются также особенности ЯП в представлении основных функций и конструкций, а также операций взаимодействия с пользователями и файлами. Следствием этого являются специфические особенности написания прикладных программ и проблемно-ориентированные области рационального применения конкретных ЯП. Для обеспечения переносимости прикладных программ формализуются конструкции интерфейсных компонентов с окружением и операционной средой.

Стандарты на языки третьего поколения прошли несколько модификаций и в настоящее время представлены в ряде действующих международных документов (табл. 1.9).

1. *Стандарты языка программирования C.* C — универсальный язык программирования высокого уровня, предназначенный для создания ПС различных классов, включая операционные системы, инструментальные ПС, а также ПС деловых и научных применений. Имеются средства структурирования данных, поддержка библиотек и административное управление памятью. Развитие языка — в обеспечении объектно-ориентированного программирования и стандартизации языка C++.

2. *Стандарты языка программирования Fortran.* Fortran — язык программирования высокого уровня, широко используемый в научно-технических применениях при интенсивных вычислительных операциях, анализе и обработке больших объемов данных.

Таблица 1.9. **Стандарты некоторых языков программирования**

Обозначение	Название стандарта
	C
ISO/IEC 9899:1999	Языки программирования. Си
ISO/IEC 9899:1999/Cor.1:2001	Языки программирования. Си. Техническая поправка 1
ISO/IEC 9899:1999/Cor.2:2004	Языки программирования. Си. Техническая поправка 2
ISO/IEC 14882:2003	Языки программирования. C++
	Fortran
ISO/IEC 1539-1:2004	Информационные технологии. Языки программирования. Фортран. Часть 1. Базовый язык
ISO/IEC 1539-1:2004/Cor.1:2006	Информационные технологии. Языки программирования. Фортран. Часть 1. Базовый язык. Техническая поправка 1
ISO/IEC 1539-2:2000	Информационные технологии. Языки программирования. Фортран. Часть 2. Строки знаков переменной длины
ISO/IEC 1539-3:1999	Информационные технологии. Языки программирования. Фортран. Часть 3. Условная компиляция
ISO/IEC TR 15580:2001	Информационные технологии. Языки программирования. Фортран. Обработка исключительных ситуаций с помощью плавающей точки
ISO/IEC TR 15581:2001	Информационные технологии. Языки программирования. Фортран. Средства типа расширенных данных
ISO/IEC TR 19767:2005	Информационные технологии. Языки программирования. Фортран. Расширенные модульные средства
	Ada
ISO/IEC 8652:1995	Информационные технологии. Языки программирования. Ada
ISO/IEC 8652:1995/Cor.1:2001	Информационные технологии. Языки программирования. Ada. Техническая поправка 1
ISO/IEC 13813:1998	Информационные технологии. Языки программирования. Настраиваемые пакеты описаний вещественных и обобщенных типов и базовых операций для языка Ada (включая векторный и матричный типы)
ISO/IEC 13814:1998	Информационные технологии. Языки программирования. Родовой пакет комплексных элементарных функций для языка Ada
ISO/IEC 15291:1999	Информационные технологии. Языки программирования. Спецификации интерфейса семантики языка Ada

Окончание табл. 1.9

Обозначение	Название стандарта
ISO/IEC 18009:1999	Информационные технологии. Языки программирования. Ada. Оценка конформности процессора языка
ISO/IEC TR 15942:2000	Информационные технологии. Языки программирования. Руководство по применению языка программирования Ada в системах высокой целостности
Pascal	
ISO/IEC 7185:1990	Инфомационные технологии. Языки программирования. Паскаль
ISO/IEC 10206:1991	Информационные технологии. Языки программирования. Расширенный Паскаль
Cobol	
ISO/IEC 1989:2002	Информационные технологии. Языки программирования. Кобол
ISO/IEC TR 19755:2003	Информационные технологии. Языки программирования, их среды и системные программные интерфейсы. Отработка объекта для языка программирования Кобол
ISO/IEC 1989:2002/Cor.1:2006	Информационные технологии. Языки программирования. Кобол. Техническая поправка 1
ISO/IEC 1989:2002/Cor.2:2006	Информационные технологии. Языки программирования. Кобол. Техническая поправка 2
Basic	
ISO/IEC 10279:1991	Информационные технологии. Языки программирования. Полный Бейсик
ISO/IEC 10279:1991/Amd.1:1994	Информационные технологии. Языки программирования. Полный Бейсик. Изменение 1
Prolog	
ISO/IEC 13211-1:1995	Информационные технологии. Языки программирования. Пролог. Часть 1. Общее ядро
ISO/IEC 13211-2:2000	Информационные технологии. Языки программирования. Пролог. Часть 2. Модули

3. *Стандарты языка программирования Ada.* Ada — наиболее мощный язык программирования сложных комплексов программ. Обеспечивает возможности строгого определения типов данных. Ориентирован на создание встроенных систем реально-

го времени, распределенных систем, высоконадежных комплексов программ и повторно используемых компонентов. Поддерживается Правительством и Министерством обороны США.

4. *Стандарты языка программирования Pascal.* Pascal — язык программирования высокого уровня, применяемый в сфере обучения студентов принципам программирования, а также в общих применениях для создания инструментального ПО.

5. *Стандарты языка программирования Cobol.* Cobol предназначен для программирования самодокументируемых деловых, коммерческих и финансовых прикладных систем с активным манипулированием данными. Является наиболее распространенным языком делового программирования. Не предназначен для создания систем реального времени, коммуникационных функций и операционных систем.

6. *Стандарты языка программирования Basic.* Basic — аббревиатура от английской фразы «многоцелевой язык символических команд для начинающих» — и в настоящее время остается самым простым языком для освоения принципов алгоритмизации и программирования. Для многих мини- и микро-ЭВМ предназначался в качестве единственного языка высокого уровня.

7. *Стандарты языка программирования Prolog.* Prolog — язык логического программирования, относится к классу декларативных языков. «Промышленный» транслятор языка, как правило, порождает исполняемый код, сопоставимый по эффективности с кодом аналогичных программ на императивных языках.

8. *Некоторые общие стандарты языков программирования* (табл. 1.10): процессоры, положения о подготовке стандартов, привязка к языкам, инвариантные типы данных.

Как правило, при создании языка выпускается частный стандарт, определяемый разработчиками языка. Если язык получает широкое распространение, то со временем появляются различные версии компиляторов, которые не точно следуют частному стандарту. В большинстве случаев идет расширение зафиксированных первоначально возможностей языка. Для приведения наиболее популярных реализаций языка в соответствие друг с другом разрабатывается согласительный стандарт. Очень важным фактором стандартизации языка программирования является своевременность появления стандарта — до широкого распространения языка и создания множества несовместимых реализаций. В процессе развития языка могут появляться новые стандарты, отражающие современные нововведения.

Таблица 1.10. **Общие стандарты ЯП**

Обозначение	Название стандарта
ISO/TR 9547:1988	Процессоры языков программирования. Методы тестирования. Руководство по их разработке и приемлемости
ISO/TR 9547:1990	Процессоры языка программирования. Методы испытаний. Руководство по разработке и применению
ISO/IEC TR 10034:1990	Руководящие положения по подготовке пунктов о согласованности в стандартах на языки программирования
ISO/IEC TR 10182:1993	Информационные технологии. Языки программирования, их контексты и системные программные интерфейсы. Руководящие положения для привязок к языкам
ISO/IEC 11404:1996	Информационные технологии. Языки программирования, их окружение и системные программные интерфейсы. Языково-инвариантные типы данных
ISO/IEC 2382-15:1999	Информационные технологии. Словарь. Часть 15. Языки программирования
ISO/IEC TR 10176:2003	Информационные технологии. Руководящие указания по подготовке стандартов на языки программирования

Игнорирование международных стандартов приводит к большим затратам при адаптации программ к другой вычислительной среде. Следовать стандартам рекомендуется разработчикам и системного, и прикладного программного обеспечения.

Упражнения

1. Назовите максимальное (минимальное) целое значение без знака (со знаком), которое можно записать в 10, 12, 14 битов.

2. Постройте блок-схемы следующих алгоритмов:
- вычисления произведения n произвольных чисел;
- вычисления суммы квадратов первых n чисел натурального ряда;
- зеркального отображения элементов двумерного массива размерностью $N \times N$ относительно побочной диагонали;
- формирования одномерного массива, каждый элемент которого представляет собой сумму элементов строки некоторого двумерного массива.

3. Модифицируйте алгоритм вычисления суммы квадратов первых n чисел натурального ряда для вычисления суммы квадратов: а) только четных чисел (до n); б) только нечетных чисел (до n).

4. Вычислите результаты выражений:
   $A*2+A >= A*A$;
   $\mathrm{Cos}(\mathrm{Abs}(X)) = \mathrm{Cos}(X)$;
   $\mathrm{Sin}(X) <= 1$;
   $\mathrm{Int}(X) <= X$;
   $X \bmod 3 > 3$;
   $A >= A \text{ div } 5 * 5$.

5. При каких значениях X значения выражений истинны:
   $\mathrm{Sin}(X) = X$;
   $X \bmod 2 = 0$;
   $X*X = X+X$;
   $\mathrm{Sqr}(X) > X$.

6. Опишите в виде массива записей информацию о студентах группы и постройте алгоритм подсчета среднего возраста студентов.

7. Разработайте процедуры, реализующие операции вставки, удаления и поиска элемента для динамического двунаправленного списка.

8. Постройте и опишите структуру данных для хранения генеалогического дерева.

9. Постройте и опишите структуру данных для хранения схемы маршрутов авиаперевозок.

10. Используя а) рекурсивную, б) итеративную схему вычислений, опишите алгоритмы вычисления суммы первых m членов ряда, член с номером n ($n = 1, 2, 3, \ldots$) которого определяется одним из следующих выражений:

$$\frac{x^n}{(2n)!};\quad \frac{(-1)^n}{n(n+1)(n+2)};\quad \frac{(-1)^{n+1}}{2n(2n+1)};\quad \frac{1}{x^3\sqrt{2n}};$$

$$\frac{(-1)^n x^n}{n!};\quad \frac{(-1)^{n+1}\sin(x)^n}{n^2}.$$

Контрольные вопросы

1. Перечислите и охарактеризуйте этапы решения задач с помощью ЭВМ на примере задачи решения квадратного уравнения $ax^2 + bx + c = 0$.
2. Охарактеризуйте базовые структуры алгоритмов.
3. Приведите примеры задач, для реализации которых применимы: а) линейные алгоритмы; б) разветвляющиеся алгоритмы; в) циклические алгоритмы.

4. Охарактеризуйте разницу между циклом типа *до* и циклом типа *пока*.
5. Приведите примеры задач, для реализации которых целесообразно применять циклические структуры: а) с постусловием; б) с предусловием.
6. Перечислите основные группы операторов языков программирования.
7. Сформулируйте назначение аппарата обработки исключений.
8. Охарактеризуйте разницу в выполнении блоков ***try-except*** и ***try-finally***.
9. Определите понятие «тип данных».
10. Охарактеризуйте разницу между простым и структурированным типом данных.
11. Перечислите типы данных, которые относятся к простым (базовым).
12. Охарактеризуйте различные способы представления числовой информации.
13. Определите тип данных *запись* и перечислите основные операции над данными этого типа.
14. Охарактеризуйте ссылочный тип данных.
15. Охарактеризуйте разницу между статическим и динамическим размещением переменных.
16. Перечислите известные линейные структуры данных и приведите их характеристики.
17. Приведите основные отличия нелинейных структур данных от линейных.
18. Перечислите известные способы организации доступа к элементам списка.
19. Опишите способ представления любой древовидной структуры в виде двоичного дерева.
20. Перечислите и охарактеризуйте типы подпрограмм.
21. Охарактеризуйте разницу между формальными и фактическими параметрами подпрограммы.
22. Охарактеризуйте разницу между вызовом параметров подпрограммы по значению и по наименованию. Приведите примеры.
23. Приведите определение рекурсивной программы (рекурсивного алгоритма).
24. Охарактеризуйте разницу между итеративной и рекурсивной схемой алгоритма. Приведите примеры.
25. Перечислите основные разделы обобщенной структуры программы на ЯВУ.
26. Определите понятия *область видимости* и *время жизни* для программного элемента.
27. Определите понятие *программный модуль*. Приведите преимущества и недостатки модульного программирования.
28. Охарактеризуйте разницу между программным модулем и компонентом.
29. Дайте определение спецификации программы.
30. Перечислите основные понятийные средства спецификации программ и охарактеризуйте их.

Глава 2
МЕТОДОЛОГИИ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Методология программирования определяет языки и системы программирования, которые будут применяться для разработки программного обеспечения. С точки зрения способа описания алгоритмов можно выделить следующие наиболее известные методологии:

- императивное программирование;
- объектно-ориентированное программирование;
- функциональное программирование;
- логическое программирование.

Основные подходы к программированию различаются по модели вычислений, положенной в основу решения задач.

Императивное (операционное, процедурное) программирование представляет собой исторически первую и наиболее распространенную методологию. При императивном подходе память вычислительной машины условно делится на две части: в одной хранятся данные в виде переменных, в другой — команды, которые изменяют содержимое переменных. Императивное программирование основано на алгоритмическом мышлении — при построении программы необходимо ясно представлять, какие действия и в какой последовательности будут проводиться при ее выполнении.

Языки, поддерживающие императивную методологию, ориентированы на запись алгоритмов — детерминированных последовательностей действий над структурами данных. Если рассматривать состояние вычислительной машины как состояние ячеек памяти, то программа, написанная на таком языке, — это последовательность операторов, изменяющих значение одной или некоторого множества ячеек.

Появление языков высокого уровня привело к заметному развитию и росту индустрии разработки программного обеспече-

ния. Вместе с этим появилась потребность в такой технологии разработки программ, которая позволила бы, во-первых, с меньшими затратами получать надежные программные продукты, и, во-вторых — поручать разработку программы коллективу программистов. При этом программа должна быть понятна всем участникам разработки, должна допускать повторное использование и модификации таким образом, чтобы изменения в тексте программы касались бы только отдельных фрагментов.

Реализация этих требований привела к появлению *структурного программирования*. Ключевая идея структурного программирования — отражение внутренней структуры алгоритма в структуре текста программы. Дальнейшее развитие идей структурного программирования — *модульное программирование*, при котором программа разбивается на модули, соответствующие функциональной декомпозиции решаемой задачи. Каждый модуль, с точки зрения его внешнего использования, реализует определенную функциональность, при этом его внутреннее устройство скрыто от того, кто его использует. Такая защищенность модуля от внешнего доступа, с одной стороны, гарантирует правильность его функционирования, а с другой стороны — предоставляет возможность вносить изменения (в случае необходимости) только в этот модуль, без изменения остальных модулей.

Дальнейшее развитие принципов инкапсуляции и наследования привело к появлению абстрактных типов данных и позднее — к идее объектно-ориентированного программирования.

Основные принципы *объектно-ориентированного программирования* (ООП) следующие:

1. Для решения исходной задачи проектируется совокупность объектов.

2. Все вычисления выполняются путем взаимодействия между объектами. Объекты взаимодействуют друг с другом посредством посылки сообщений. Сообщение инициирует выполнение действия. Сообщение может иметь набор аргументов, необходимых для выполнения действия.

3. Каждый объект имеет независимую память.

4. Каждый объект является представителем класса, который выражает общие свойства объектов и их поведение.

5. Классы организованы в единую иерархическую древовидную структуру с общим корнем. Память и поведение объектов некоторого класса наследуются объектами класса, расположенного ниже в иерархическом дереве.

Языки объектно-ориентированного программирования основаны на построении объектов как набора данных и операций над ними.

Непроцедурное (декларативное) программирование появилось в начале 70-х годов, но стремительное его развитие началось в 80-е годы, когда был разработан японский проект создания ЭВМ пятого поколения. К непроцедурному программированию относятся функциональное и логическое программирование.

В *функциональном программировании* программа рассматривается как суперпозиция функций, применяемых к начальным данным для получения требуемого результата. Один из основных вычислительных приемов — рекурсия, т. е. вычисление значения функции через значение этой же функции от других аргументов.

Языки функционального программирования не поддерживают понятие переменной, как ячейки памяти, не объявляют изменяемые объекты и, таким образом, не содержат оператора присваивания. В таких языках отсутствуют операторы циклов. Запись программы на языке функций ближе к записи математического решения задачи.

Логическое программирование основано на формальной логике. Программа задается как набор логических утверждений: формулируется база знаний задачи — набор логических аксиом («факты») и правил вывода («правила»). Запросы к базе знаний формулируются в виде целевых утверждений («целей»). Выполнение программы состоит в доказательстве целевого утверждения для объявленной базы знаний. Логическая система находит решение задачи путем применения операций унификации (сопоставления) и редукции (преобразования, упрощения).

Языки логического программирования (или языки системы правил) основываются на определении набора фактов и правил логического вывода. В таких языках отсутствует понятие глобальной переменной и оператора присваивания. Правила могут задаваться в виде утверждений и в виде таблиц решений.

Далее будут рассмотрены представленные выше методологии программирования и приведено описание синтаксиса и семантики одного языка программирования для каждой из методологий. Синтаксис языка определит систему правил написания различных языковых конструкций, а семантика — смысл этих конструкций. Так как семантика языка взаимосвязана с используемой вычислительной моделью, будет охарактеризована вычислительная модель каждой из методологий.

2.1. Императивное программирование

Императивное (процедурное) программирование — исторически первая технология программирования. Она ориентирована на классическую фон-неймановскую архитектуру[1] и поддерживается концепцией алгоритма — принципом последовательного изменения состояния вычислителя пошаговым образом. При этом управление изменениями полностью определено и полностью контролируемо. Таким образом, процедурное программирование основано на алгоритмическом мышлении и может служить средством его развития.

Императивное программирование наиболее пригодно для решения задач, в которых последовательное исполнение каких-либо команд является естественным. Поскольку практически все современные компьютеры императивны, эта технология позволяет порождать достаточно эффективный исполняемый код, однако с ростом сложности задачи императивные программы становятся все менее читаемыми.

Основные структурные элементы языков, поддерживающих императивное программирование — переменные, константы, операторы присваивания, циклы, условные операторы, процедуры и функции.

---

[1] Принципы фон Неймана:

- оперативная память организована как совокупность *машинных слов* (МС) фиксированной длины или *разрядности* (имеется в виду количество двоичных единиц или *бит*, содержащихся в каждом МС);
- ОП образует единое *адресное пространство*, адреса МС возрастают от *младших к старшим*;
- в ОП размещаются как данные, так и программы, причем в области данных *одно слово*, как правило, соответствует *одному числу*, а в области программы — *одной команде* (машинной *инструкции* — минимальному и неделимому элементу программы);
- команды выполняются в *естественной последовательности* (по возрастанию адресов в ОП), если/пока не встретится *команда управления* (условного/безусловного перехода), в результате которой естественная последовательность нарушится;
- центральный процессор (ЦП) может *произвольно* обращаться к любым адресам в ОП для выборки и/или записи в МС чисел или команд.

2.1.1. Вычислительная модель

В основе императивного программирования лежит описание последовательного изменения состояний вычислителя. Состояние ЭВМ как вычислителя характеризуют:

- два типа данных: адреса и значения;
- конечный набор базовых операций (система команд);
- единственная структура данных — кортеж ячеек (пар адрес—значение) с линейно упорядоченными адресами;
- регистр команд (выделенная ячейка, в которой хранится адрес подлежащей выполнению команды).

Реализация нелинейных структур ориентирована на использование указателей, объединяющих линейные фрагменты.

Математической моделью императивного программирования служит машина Тьюринга — абстрактное вычислительное устройство, предложенное в конце 30-х годов XX в. Аланом Тьюрингом для формального определения алгоритма.

Машина Тьюринга

Классическая *машина Тьюринга* состоит из трех компонентов (рис. 2.1):

- бесконечной (в обе стороны) ленты, разбитой на ячейки, которые могут содержать строго по одному символу из некоторого конечного алфавита;
- конечного автомата, представляющего собой устройство, находящееся в каждый дискретный момент времени в некотором состоянии;
- сканирующей головки, которая за единичный момент времени сканирует одну ячейку ленты и в зависимости от состояния автомата может изменять содержимое ячейки, сдвигаться на одну ячейку вправо или влево и переводить автомат в другое состояние.

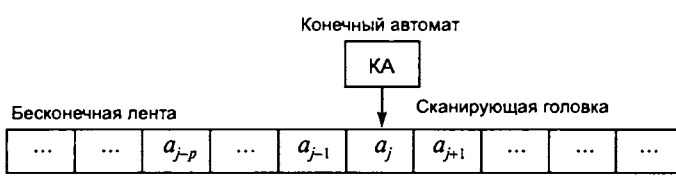


Рис. 2.1. Принципиальная схема абстрактной машины Тьюринга

Алфавит любой машины Тьюринга всегда содержит символ Λ, идентифицирующий пустую ячейку ленты: $A = \{\Lambda, a_1, a_2, ..., a_n\}$.

Множество состояний конечного автомата — $Q = \{!, q_1, q_2, ..., q_n\}$, где !-состояние определяет останов. Когда машина переходит в состояние останова, она заканчивает вычисления.

Несмотря на свою простоту, машина Тьюринга может выполнить любое вычисление, которое доступно современному компьютеру.

Действие, которое будет выполнять машина Тьюринга, зависит только от состояния машины и символа в ячейке, над которой находится сканирующая головка машины. На основе этой информации (состояния и символа под головкой) машина Тьюринга может выполнить одно из трех действий:

- записать любой символ из алфавита A в ячейку;
- передвинуться на одну ячейку вправо или влево (или остаться на месте);
- установить состояние конечного автомата (любое из множества состояний Q).

Машина Тьюринга работает согласно таблице правил или программе, которая представляет собой описание множества состояний Q в зависимости от символов алфавита A, прочитанных с ленты. На рис. 2.2 представлена программа, прибавляющая к целому числу единицу. Алфавит машины Тьюринга состоит из десятичных цифр: $A = \{\Lambda, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Исходное положение сканирующей головки — первая цифра числа. Состояние q_1 перемещает последовательно сканирующую головку направо до первой после числа пустой ячейки, после этого автомат переходит в состояние q_2. В состоянии q_2 сканирующая головка увеличивает на 1 любую, кроме 9, цифру и останавливается. На цифре 9 происходит запись в ячейку цифры 0 и перемещение сканирующей головки влево. Появление нового старшего разряда числа фиксируется записью цифры 1 в первую пустую ячейку ленты слева от числа.

	Λ	0	1	2	3	4	5	6	7	8	9
q_1	Лq_2	Пq_1	Пq_1	Пq_1	Пq_1	Пq_1	Пq_1	Пq_1	Пq_1	Пq_1	Пq_1
q_2	1!	1!	2!	3!	4!	5!	6!	7!	8!	9!	0Лq_2

Рис. 2.2. Программа прибавления 1 к целому числу

Одна из характерных черт императивного программирования — наличие переменных и оператора «разрушающего присваивания». Это означает, что каждый оператор присваивания некоторой переменной X некоторого нового значения V уничтожает то значение, которое содержала переменная до выполнения оператора.

Структурное императивное программирование

Создателем структурного подхода считается Э. Дейкстра [14]. Структурный подход в программировании (см. гл. 1, п. 1.7) базируется на двух основополагающих принципах:

- использование процедурного стиля программирования;
- последовательная декомпозиция алгоритма решения задачи сверху вниз.

Структурное программирование — традиционная технология программирования 1970-х годов, основанная на принципах модульного нисходящего программирования и структурного проектирования данных и процедур (так разываемое «программирование без `goto`»).

Технология структурного программирования, как важнейшее развитие императивной модели, определила значение модульного построения программ при разработке больших проектов, однако в языках программирования единственным способом структуризации программы оставался способ ее составления из подпрограмм и функций.

2.1.2. Синтаксис и семантика языков императивного программирования

Основным синтаксическим понятием в языках, поддерживающих технологию императивного программирования, является оператор. Состав операторов языка строится из множества типовых операторов управления вычислительным процессом (см. п. 1.2).

Операторы делятся на атомарные (например, оператор присваивания, оператор безусловного перехода, оператор вызова процедуры и т. п.) и структурные, — объединяющие другие операторы в новый, более крупный оператор (например, оператор выбора, оператор цикла, составной оператор и т. п.).

Для описания синтаксиса языков в основном используются средства, которые позволяют одни синтаксические категории последовательно определять через другие, как, например, формальная грамматика Бэкуса-Наура и синтаксические диаграммы (см. гл. 1, п. 1.8). При этом многие определения носят рекурсивный характер.

Приведем фрагмент описания синтаксиса некоторого языка, поддерживающего императивное программирование:

```
<оператор> ::= <простой оператор>; | <структурный оператор>;
<простой оператор> ::= <оператор присваивания> | <оператор вызова > |
                       <оператор безусловного перехода>
<оператор присваивания> ::= <переменная> := <выражение>
<оператор вызова> ::= <имя подпрограммы> [(<список параметров>)]
<имя подпрограммы> ::= <идентификатор>
<идентификатор> ::= <буква>|<идентификатор><буква>|
                    <идентификатор><цифра>
<список параметров> ::= <параметр>|<список параметров>, <параметр>
<оператор безусловного перехода > ::= goto <идентификатор >
<структурный оператор> ::= <составной оператор> |<оператор ветвления> |
                           <оператор цикла>
<составной оператор > ::= begin <последовательность операторов> end
<последовательность операторов> ::= <оператор> |
                    <последовательность операторов>; <оператор>
<оператор выбора> ::= if <выражение> then <оператор> |
                      if <выражение> then <оператор> else <оператор>
<оператор цикла> ::= while <выражение> do <оператор> |
  for <идентификатор> := <выражение> to <выражение> do <оператор>
```

Операторы в программе исполняются в порядке, определенном алгоритмом решения задачи. Выполнение простых операторов приводит к соответствующему единичному изменению состояния вычислителя. Составные операторы задают некоторую последовательность действий. Операторы, входящие в оператор выбора, исполняются или не исполняются в зависимости от значения логического условия.

В табл. 2.1 дается обзор основных операторных возможностей ЯП: Pascal, Basic и C (Си).

Традиционное средство структурирования в языках императивного программирования — подпрограмма (процедура или функция). Подпрограмма может иметь параметры и содержать в своем описании локальные определения. При вызове подпрограмм может быть использована рекурсия (см. п. 1.5 гл. 1). Функции возвращают значения как результат своей работы.

Таблица 2.1. **Основные структуры некоторых ЯП**

Элемент языка	Pascal	Basic	C
Пересылка, присваивание	`:=`	`=`	`=`
Переход на метку	`goto Label`	`GoTo Label`	`goto Label`
Ветвление	`if...then...else` `case`	`If...Then...Else` `Select Case`	`if ()...else` `switch`

Циклы	`for ...do`
	`repeat...until`
	`while...do`
		`For ...Next`
		`Do...Loop`
		`While...Wend`
			`for()`
			`do... while ()`
			`while ()`

Переход к заголовку цикла	`continue`		`continue`
Выход из цикла во вне	`Break`	`Exit For` `Exit Do`	`break`
Блоки, составные операторы	`begin... end`		`{}`

Комментарии	`(*текст*)`
	`{ текст }`
	`// текст`
		`REM текст`
		`'текст`
			`/* текст */`

Большинство наиболее известных и распространенных императивных языков программирования были созданы в конце 50-х — середине 70-х годов XX в.:

- Fortran (1954 г.);
- Algol (1960 г.);
- Basic (1964 г.);
- Pascal (1970 г.);
- C (1972 г.).

В гл. 4 настоящего пособия приведено описание ЯВУ Object Pascal, который является объектно-ориентированным расширением императивнного языка Pascal (с сохранением императивной семантики).

2.2. Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — подход, использующий объектную декомпозицию, основанную на выделении объектов и связей между ними. В отличие от процедурного подхода к программированию, когда описание алго-

ритма решения некоторой задачи представляет собой последовательность действий, объектно-ориентированный подход предлагает описывать программные системы в виде взаимодействия объектов и, прежде всего, создать некоторый инструментарий, присущий решаемой задаче, а затем уже программировать в терминах этой задачи.

Объектно-ориентированное программирование является мощным средством для моделирования отношений между объектами практически в любой предметной области. Особенно удобно и легко в объектах выразить взаимодействие между различными элементами графического интерфейса пользователя.

2.2.1. Вычислительная модель

Динамика поведения объектно-ориентированной системы описывается в терминах обмена сообщениями между объектами. «Чистое» объектно-ориентированное программирование поддерживает всего одну операцию: послать объекту *O* сообщение *Mes* с параметрами P_1, ..., P_n. И само сообщение, и его параметры — это, в свою очередь, объекты, которые могут быть результатами обработки других сообщений.

Объекты и классы

В объектно-ориентированном программировании базовыми единицами являются объекты.

Объект — это понятие, сочетающее в себе совокупность данных и действий над ними. Каждый объект характеризуется своим состоянием, которое определяется текущими значениями его *атрибутов* (характеристических свойств). Атрибутами объекта могут быть не только атомарные величины (числа, символы, логические значения), но и объекты. Все атрибуты объекта делятся на две группы — «внутренние» атрибуты (те, которые доступны только самому объекту) и атрибуты, входящие в интерфейс объекта (их часто называют *свойствами*).

Интерфейс объекта — это описание того, как он может взаимодействовать с окружающим миром. Интерфейс объекта определен полностью его свойствами и *методами*. Методы — это действия над данными объекта.

Объект сохраняет свое состояние до тех пор, пока оно не будет изменено через вызов методов этого объекта. Это существен-

но ограничивает возможность несанкционированного или некорректного доступа к объекту. Управление состояниями объекта через вызов методов в конечном итоге определяет поведение объекта.

Объект может посылать *сообщения* другим объектам и принимать сообщения от них. Сообщение является совокупностью данных определенного типа, передаваемых объектом-отправителем объекту-получателю, имя которого указывается в сообщении. Получатель реагирует на сообщение выполнением некоторого метода.

Таким образом, совокупность объектов образует среду, в которой вычисления выполняются путем обмена сообщениями между объектами. Состояние вычислительной среды в ООП характеризуется совокупным состоянием объектов, что в принципе отличает объектно-ориентированные вычисления от вычислений, заданных на процедурно-ориентированных языках.

Объекты с одинаковыми возможностями (данными и методами) объединяет термин *класс*. Класс описывает общее поведение и характеристики набора аналогичных друг другу объектов. *Объект — это экземпляр класса* или, другими словами, переменная, тип которой задается классом. Объекты в отличие от классов реальны, т. е. существуют и хранятся в памяти во время выполнения программы. Соотношения между объектом и классом аналогичны соотношениям между переменной и типом.

Объектно-ориентированное программирование позволяет программировать в терминах классов: определять классы, конструировать производные классы (подклассы) на основе существующих классов, создавать объекты, принадлежащие классу (экземпляры класса). Каждый класс задается своим описанием на языке ООП, которое включает информацию, необходимую для создания объектов данного класса и организации работы с ними.

У каждого объекта есть ссылка на класс, к которому он относится. При приеме сообщения объект обращается к классу для обработки данного сообщения. Если сам класс не располагает методом для обработки сообщения, оно может быть передано вверх по иерархии наследования.

Так как объекты взаимодействуют исключительно за счет обмена сообщениями, они могут ничего не знать о реализации обработчиков сообщений друг у друга. Таким образом, взаимодействие описывается исключительно в терминах сообщений или *событий*, которые достаточно легко привязать к конкретной

предметной области. В этом состоит свойство *абстракции* (т. е. описания взаимодействия исключительно в терминах предметной области).

Свойства ООП

Структуру и поведение объектов в ООП определяют свойства инкапсуляции, наследования и полиморфизма.

Инкапсуляция. Объединение всех данных и методов объекта (включая данные и методы объектов-предков) называется *инкапсуляцией.* Механизм инкапсуляции скрывает данные и методы объекта от внешнего вмешательства или неправильного использования, облегчая тем самым понимание работы программы, а также ее отладку и модификацию, так как только в очень редких случаях разработчика интересует внутренняя реализация объектов — главное, чтобы объект обеспечивал функции, которые он должен предоставить.

Взаимодействие с объектом происходит через интерфейс. Только содержимое интерфейса предоставляется данным объектом в распоряжение других объектов, благодаря этому предотвращается доступ к внутренним переменным объекта. Таким образом, инкапсуляция обеспечивает использование объекта без знания его реализации.

Данные и методы внутри объекта могут обладать различной степенью открытости (или доступности). Некоторые из них объявляются общедоступными, другие доступны только из методов самого объекта. Обычно открытые данные используются для того, чтобы обеспечить контролируемый интерфейс с его закрытой частью.

Наследование. *Наследование* позволяет определять новые классы в терминах существующих классов и повторно использовать уже созданную часть программного кода в других проектах. Посредством наследования формируются связи между объектами, а для выражения процесса наследования используют понятия «родители» и «потомки». В программировании наследование служит для сокращения избыточности кода, и суть его заключается в том, что уже существующий интерфейс вместе с его программной частью можно использовать для других объектов. Точнее, объект может наследовать свойства другого объекта и добавлять к ним черты, характерные только для него. При наследовании могут также проводиться изменения интерфейсов.

Обычно, если объекты соответствуют конкретным сущностям реального мира, то классы являются абстракциями, выступающими в роли понятий. Между классами как между понятиями существуют иерархические отношения, связывающие класс с его родителем и потомком, которые и реализуются механизмом наследования.

Наследование выполняет в ООП несколько важных функций:

- моделирует концептуальную структуру предметной области (в виде иерархии классов);
- позволяет использовать одни и те же писания для задания разных классов;
- обеспечивает программирование больших систем путем пошаговой конкретизации классов.

Обычно в объектно-ориентированных языках существует класс, являющийся вершиной всей иерархии наследования (как правило, называемый `Object`), а одним из свойств объекта является ссылка на объект-предок, которому переадресуются все сообщения, не обрабатываемые данным объектом.

Полиморфизм. Полиморфизмом называется способность различных объектов по-разному обрабатывать одинаковые сообщения. При этом различные объекты используют одинаковую абстракцию, т. е. могут обладать свойствами и методами с одинаковыми именами. Однако обращение к ним будет вызывать различную реакцию для различных объектов. На практике это означает, что можно создать общий интерфейс вызова для группы близких по смыслу действий.

Проиллюстрируем свойства ООП на примере решения задачи размещения и перемещения некоторого количества мерцающих разноцветных точек на плоскости экрана.

Точка на экране характеризуется координатами `X`, `Y`, имеет цвет, может быть видимой или невидимой и может перемещаться по экрану.

Очевидно, что основой изображения точки является ее положение (позиция) на экране (например, значения координат `X` и `Y` относительно левого верхнего угла экрана). Таким образом, может быть объявлен класс `Позиция` со свойствами — координатами `X` и `Y`, имеющими тип целое число, и методом `НазначитьXY` (назначить координаты):

```
Позиция(X, Y, НазначитьXY)
```

Далее объявим класс Точка, который может быть описан следующим образом:

```
Точка(X, Y, Видимость, Цвет, НазначитьXY, Назначить цвет,
 Зажечь, Погасить, Переместить)
```

При таком объявлении свойства и методы класса Позиция полностью входят в класс Точка. Используя механизм наследования, опишем класс Точка как потомка класса Позиция:

```
Точка(Позиция, Видимость, Цвет, НазначитьЦвет, Зажечь,
 Погасить, Переместить)
```

Интерфейс такого объекта составляют только методы. Атрибуты X, Y, Видимость, Цвет получают конкретные значения при вызове интерфейсных методов НазначитьXY, Переместить (для атрибутов X, Y), НазначитьЦвет (для атрибута Цвет), Зажечь, Погасить (для атрибут Видимость). Тогда для создания объекта такого класса (например, Точка1) необходимо активизировать процедуру размещения объекта и присвоить атрибутам конкретные значения, активизировав соответствующие методы, например:

```
Точка1 = Создать.Точка;
Точка1.НазначитьXY(3,5);
Точка1.НазначитьЦвет(Красный);
Точка1.Зажечь;
```

В результате таких действий на экране появится красная точка с координатами X = 3, Y = 5.

Чтобы проиллюстрировать свойство полиморфизма, объявим класс цветных кругов, которые задаются координатой центра и радиусом. Этот класс должен наследовать все возможности класса Точка, но методы НазначитьЦвет, Зажечь, Погасить, Переместить, по результату действия являясь теми же самыми, фактически не могут быть реализованы одинаковой последовательностью команд в случае с точкой и кругом. Эта ситуация разрешается путем использования полиморфизма: методы классов имеют одинаковое имя (и, соответственно, одинаковый ожидаемый результат), а внутреннее описание методов будет различно для разных классов. Тогда класс Круг может иметь следующее описание:

```
Круг(Точка, Радиус, Назначить цвет, Зажечь,
Погасить, Переместить)
```

Компоненты

На логическом уровне компонент — это объект, объединяющий состояние и интерфейс. Физически компонент (в соответствии с компонентной концепцией разработки программ) представляет собой исполняемый программный модуль.

Фирмой Microsoft (с целью стандартизации программных компонентов) была разработана технология ActiveX, в основе которой лежит COM (Component Object Model) — модель компонентного объекта. Эта системная технология объединяет совокупность средств, с помощью которых объекты, разработанные различными разработчиками на разных языках программирования и работающие в разных средах, могут взаимодействовать друг с другом без какой-либо модификации их исполняемых модулей. Сутью данной технологии является то, что программы строятся из компонентов, представляющих собой объекты. При этом компоненты — непосредственно исполняемые файлы, и тем самым не связаны с языком программирования. Компонент (для обеспечения работы с ним) достаточно зарегистрировать в операционной системе и тогда он будет доступен любой программе, исполняющейся на данной ЭВМ.

У компонента имеются два типа интерфейсов: интерфейс стадии проектирования и интерфейс стадии выполнения. Интерфейс проектирования позволяет включать компоненты в современные среды разработки приложений, а интерфейс выполнения управляет работой компонента во время выполнения программы.

Разработка любого приложения состоит из двух взаимосвязанных этапов:

- проектирование и создание функционального интерфейса приложения (т. е. набора визуальных компонентов, которые будут обеспечивать взаимодействие пользователя и вычислительной среды);
- программирование процедур обработки событий, возникающих при работе пользователя с приложением.

На *первом этапе* (т. е., на этапе проектирования интерфейса — формирования общего вида главного окна при выполнении приложения и способов управления работой приложения) для каждого компонента необходимо определить его внешний вид, размеры, способ и место размещения в области окна приложе-

ния (т. е. реализовать интерфейс разработки и интерфейс выполнения).

С точки зрения внутренней структуры компоненты разбиваются на три группы. На рис. 2.3 представлена графическая интерпретация этого разбиения.

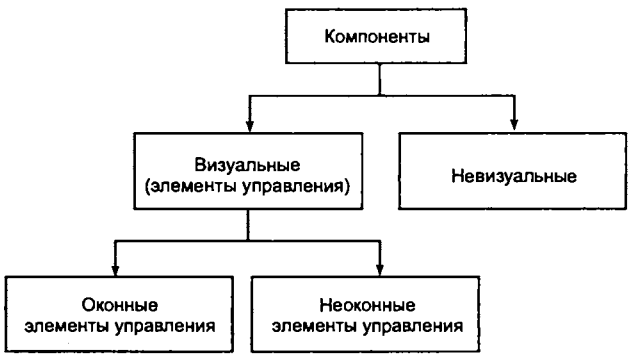


Рис. 2.3. Иерархия групп компонентов схожей внутренней структуры

Визуальные компоненты (элементы управления) характеризуются наличием свойств размеров и положения в области окна и на стадии разработки приложения обычно находятся на форме в том же месте, что и во время выполнения приложения (например, кнопки, списки, переключатели, надписи). Визуальные компоненты имеют две разновидности — «оконные» и «не оконные» (графические):

- «оконные» визуальные компоненты (самая многочисленная группа компонентов) — это компоненты, которые могут получать фокус ввода (т. е. становиться активными для взаимодействия с пользователем) и содержать другие визуальные компоненты;
- «неоконные» (графические) визуальные компоненты не могут получать фокус и содержать другие визуальные компоненты (например, надписи и графические кнопки).

Невизуальные компоненты на стадии разработки не имеют своего фиксированного местоположения и размеров. Во время выполнения приложения некоторые из них иногда становятся видимыми (например, стандартные диалоговые окна открытия и сохранения файлов), а другие остаются невидимыми всегда (например, таблицы базы данных).

Важной характеристикой компонента, как и любого объекта, являются его *свойства* — атрибуты, определяющие его состояние и поведение. Различают три типа свойств компонента.

Первые — свойства *времени проектирования*. Установленные для них значения будут использоваться в момент первого отображения компонента и в дальнейшем могут быть изменены во время выполнения приложения.

Вторые — *динамические* свойства. Изменением их значений можно управлять только изнутри программного кода (во время выполнения приложения).

Третьи — так называемые свойства *только-для-чтения*, которые могут быть прочитаны и использованы при выполнении программы, но не могут быть изменены.

Второй этап — непосредственное программирование процедур обработки событий, исходящих от компонентов. Основная задача при разработке таких процедур — запрограммировать реакцию на все возможные изменения состояний объектов.

Объектно-ориентированные языки программирования

Объектно-ориентированные языки программирования обязательно содержат конструкции, позволяющие объявлять классы, создавать и уничтожать объекты, принадлежащие классам и т. п. Объектно-ориентированные языки можно разделить на две группы:

- «чистые» объектно-ориентированные языки, в наиболее классическом виде поддерживающие объектно-ориентированную технологию:
  - Simula (1962 г.);
  - Smalltalk (1972 г.);
  - Beta (1975 г.);
  - Cecil (1992 г.);
  - Java (1995 г.);
- «гибридные» языки, появившиеся в результате внедрения объектно-ориентированных конструкций в популярный императивный язык программирования:
  - Object Pascal (1984 г.);
  - C++ (1983 г.).

Первым «настоящим» языком ООП принято считать Smalltalk, который был разработан в Паоло-Альто, в лаборатории компании Ксерокс.

2.2.2. Язык объектно-ориентированного программирования Java

Язык Java был разработан в компании Sun Microsystems в 80-х — 90-х годах XX в. Цель разработки — создание объектно-ориентированного машинно-независимого языка программирования для Internet.

Программирование для сети требовало создания унифицированного языка, одинаково интерпретируемого на различных аппаратных платформах. Первоначально в качестве такого языка создатели Java хотели использовать C++ путем адаптации к требованиям работы в сети, расширения его возможностей и преодоления его недостатков (основным недостатком было явное распределение памяти и, соответственно, работа с указателями, что затрудняло использование программ на различных аппаратных платформах).

Но реально Java не стал «улучшенным C++»: объектная модель языка существенно отличается от объектной модели C++. Java представляет собой объектно-ориентированный машинно-независимый синтаксически схожий с C++ язык программирования в сети. Программа, написанная на языке Java, не зависит от архитектуры процессора и может выполняться без предварительной перекомпиляции в любой операционной системе на любом типе компьютера, для которого существует *виртуальная Java-машина* (*Java Virtual Machine*). Java-машина при этом выполняет роль промежуточного звена между компилятором и компьютером. Java-компилятор транслирует исходный текст программы в низкоуровневые байт-коды, одинаковые для всех Java-машин. Каждая команда байт-кода состоит из однобайтного кода операции и одного или нескольких аргументов. В процессе выполнения программы виртуальная Java-машина читает и интерпретирует эти байт-коды, в результате этого Java-программа может выполняться на любом типе компьютера.

Лексика языка программирования Java

При записи текстов программ на языке Java используются заглавные и строчные буквы латинского алфавита (A B C D E F G H I J K L N O P Q R S T U V W X Y Z a b c d e f g h i j k l n o p q r s t u v w x y z), набор символов Unicode, цифры (0 1 2 3 4 5 6 7 8 9) и ограничители.

В наборе символов Unicode определено 34 168 символов. Символ Unicode описывается 16-разрядным значением, например:

`\u0660-\u0669` — арабские цифры;

`\u0024` — знак доллара («$»).

Ограничители представляют собой знаки пунктуации, знаки операций, разделители и служебные (зарезервированные) слова.

Язык Java рассматривает одну и ту же малую и большую буквы как различные символы.

Escape-последовательности. Пробельные или неграфические символы представляются специальными символьными комбинациями — escape-последовательностями. Они служат для спецификации таких действий, как возврат каретки и табуляция, а также для задания символьных представлений некоторых кодов. Escape-последовательность состоит из наклонной черты влево, за которой следует буква, знаки пунктуации или комбинация цифр. В табл. 2.2 приведен список escape-последовательностей языка.

Таблица 2.2. **Escape-последовательности**

Escape-последовательность	Наименование символа
`\n`	Новая строка
`\t`	Горизонтальная табуляция
`\b`	Возврат назад (`back space`)
`\r`	Возврат каретки
`\f`	Конец страницы
`\'`	Одиночная кавычка
`\"`	Двойная кавычка
`\\`	Наклонная черта влево
`\ddd`	ASCII символ в восьмеричном представлении (0ddd)
`\xdd`	ASCII символ в шестнадцатеричном представлении (0xdd)
`\udddd`	Unicode (0xdddd)

Если наклонная черта влево предшествует символу, не включенному в этот список, то наклонная черта влево игнорируется.

Например, последовательность `\c` представляет символ «c» в символьной строке или константе-символе.

Последовательности `\ddd` и `\xdd` позволяют задать любой символ в ASCII как последовательность от одной до трех восьмеричных цифр или от одной до двух шестнадцатеричных цифр. Например, символ пробела может быть задан как `\040`, `\40`, `\x20`. Код ASCII «ноль» может быть задан как `\0` или `\x0`.

Зарезервированные слова языка Java:

abstract	boolean	break	byte	case	cast	catch
char	class	const	continue	default	do	double
else	extends	final	finally	float	for	future
generic	goto	if	implements	import	inner	instanceof
int	interface	long	native	new	null	operator
outer	package	private	protected	public	rest	return
short	static	super	switch	syncronized	this	throw
throws	transien	try	var	void	volatile	while

Идентификаторы. Идентификатором является любая неограниченная последовательность символов Unicode (выше шестнадцатеричного `0C00`), ASCII-символов и цифр, начинающаяся с символа подчеркивания, символа доллара или буквы (символа Unicode).

Переменные и типы данных

В языке используются следующие простые (базовые) типы данных:

- целый;
- вещественный;
- логический;
- символьный.

Целый тип данных. Целый тип данных представлен множеством типов: `byte`, `short`, `int`, `long` (табл. 2.3). Точность вычислений сохраняется независимо от платформы.

Вещественный тип данных. Представлен типами обычной и двойной точности: `float`, `double` (табл. 2.3).

Логический тип данных. Тип `boolean` принимает одно из двух значений — `true` или `false`. Значения входят в список зарезервированных слов языка Java.

Таблица 2.3. **Базовые типы данных Java**

Название типа	Размер памяти (в байтах)	Область изменения
`byte`	1	–128..127
`short`	2	–32 768 ..32 767
`int`	4	$-2^{31}..2^{31}-1$
`long`	8	$-2^{63}..-2^{63}-1$
`float`	4	Число с плавающей точкой обычной точности: максимальное абсолютное значение мантиссы = 2^{24}, значение экспоненты от –149 до 104
`double`	8	Число с плавающей точкой двойной точности: максимальное абсолютное значение мантиссы = 2^{53}, значение экспоненты от –1045 до 1000
`boolean`	1	`true .. false`
`char`	2	0 .. 65535

Символьный тип данных. Тип `char` является беззнаковым целым длиной 2 байта и может принимать значения от 0 до 65 535. Таким образом, он покрывает все символы Unicod и, кроме того, имеет резерв на будущее.

Переменные и константы. Переменные, объявленные как переменные одного из простых типов, всегда содержат значение простого типа. Значение такой переменной может быть изменено только операторами, использующими эту переменную.

Константы могут представляться значениями простых типов или строкой. Возможны следующие типы констант:

- целые;
- с плавающей точкой;
- логические;
- символьные;
- строковые.

Целые константы могут быть десятичными, шестнадцатеричными или восьмеричными и задаваться символами ASCII или символами Unicode. Целая константа может дополнительно иметь тип `long`, указываемый суффиксом `L`.

Десятичная константа начинается с любой цифры от 1 до 9.

Шестнадцатеричная константа начинается с символов `0x` или `0X` и может далее содержать цифры от 0 до 9 и буквы от `a` до `f`.

Восьмеричная константа начинается с цифры 0 и может далее содержать цифры от 0 до 7.

Таким образом, число 100 можно представить одним из следующих способов:

```
int o_int = 0144;   // В восьмеричном формате
int d_int = 100;    // В десятичном формате
int h_int = 0x64;   // В шестнадцатичном формате
int l_int = 100L;   // Длинное десятичное целое
```

Константы с плавающей точкой *можно записывать как с указанием экспоненты, так и с указанием десятичной точки, например*:

```
37.55 или 3.755E1
```

По умолчанию константа с плавающей точкой имеет тип `double`. Зафиксировать тип константы можно с помощью суффикса `D` (`double`) или `F` (`float`): `3.755E1F` — константа типа `float`.

Логические константы могут иметь одно из значений логического типа — `true` или `false`.

Символьные константы могут содержать печатный символ или символ, определяемый escape-последовательностью. Символ заключается при этом в одинарные кавычки:

```
char a = 'b' // Символьная константа 'b'
```

Строковая константа состоит из нуля или более символов, заключенных в двойные кавычки. Каждая строковая константа ссылается на экземпляр класса `String`. Отметим, что язык Java не имеет типа `String` и любая строка реализуется как объект.

Операции

Все операции языка подразделяются на следующие группы:

- арифметические;
- операции отношения;
- логические;
- операции с битами информации.

Каждой группе операций соответствуют определенные типы переменных и констант.

В табл. 2.4 представлен список операций. Символ «У» в столбце «Вид» определяет операцию как унарную, символ «Б» — как

бинарную. Операции должны использоваться точно так, как они представлены в таблице: без пробельных символов внутри последовательности знаков, изображающих операцию.

Таблица 2.4. **Операции язык Java**

Операция	Вид	Наименование	Типы операндов	Тип результата
		Арифметические		
-	У	Арифметическое отрицание	Целый Вещественный	Целый Вещественный
++	У	Инкремент	Целый	Целый
--	У	Декремент	Целый	Целый
+	Б	Сложение	Целый Вещественный	Целый Вещественный
-	Б	Вычитание	Целый Вещественный	Целый Вещественный
*	Б	Умножение	Целый Вещественный	Целый Вещественный
/	Б	Деление	Целый Вещественный	Целый Вещественный
%	Б	Модуль (остаток)	Целый Вещественный	Целый Вещественный
		Операции отношения		
<	Б	Меньше	Целый Вещественный	Логический
<=	Б	Меньше или равно	Целый Вещественный	Логический
>	Б	Больше	Целый Вещественный	Логический
>=	Б	Больше или равно	Целый Вещественный	Логический
==	Б	Равно	Целый Вещественный	Логический
!=	Б	Не равно	Целый Вещественный	Логический
		Логические		
!	У	Логическое НЕ	Логический	Логический

Окончание табл. 2.4

Операция	Вид	Наименование	Типы операндов	Тип результата
||	Б	Логическое ИЛИ	Логический	Логический
&&	Б	Логическое И	Логический	Логический
Операции с битами информации				
~	У	Побитовое дополнение	Целый	Целый
<<	Б	Сдвиг влево	Целый	Целый
>>	Б	Сдвиг вправо	Целый	Целый
>>>	Б	Сдвиг вправо с заполнением нулями	Целый	Целый
&	Б	Побитовое И	Целый	Целый
|	Б	Побитовое ИЛИ	Целый	Целый

В Java существуют версии сокращенной записи для каждой из бинарных операций, работающих с целыми и вещественными типами данных. Синтаксически это семейство дополнительных кодирований операций выглядит следующим образом: `<операция>=`, где `<операция>` — это изображение любой из допустимых операций. Смысл такого сокращения в объединении операции с оператором присваивания значения. Например, следующие две записи операции присваивания эквивалентны:

```
i_current = i_current + 10;
i_current += 10;
```

Операторы языка

Все операторы языка Java можно разделить на следующие группы:

- пустой оператор;
- помеченный оператор;
- оператор присваивания;
- условные операторы;
- операторы цикла;
- оператор перехода;
- операторы исключений;
- блок;
- прочие операторы.

Пустой оператор. Пустой оператор не выполняет никаких действий и обозначается символом точка с запятой («;»).

Помеченный оператор. Помеченным оператором может быть любой оператор, перед которым стоит идентификатор метки и символ двоеточия.

Оператор присваивания. Оператор предназначен для присваивания переменной заданного значения.

Синтаксис:

```
<идентификатор> = <выражение>;
```

Тип данных переменной, имя которой стоит в левой части оператора присваивания, должен быть совместим с типом данных выражения.

Условные операторы. К условным операторам относятся операторы `if` и `switch`.

Синтаксис оператора `if`:

```
if (<выражение>)
  <оператор 1>
[else
  <оператор 2>]
```

Тело оператора `if` выполняется селективно, в зависимости от значения выражения. Сначала вычисляется выражение. Если значение выражения истина, то выполняется оператор <оператор 1>. Если выражение ложно, то выполняется оператор <оператор 2>, непосредственно следующий за ключевым словом `else`. Если <выражение> ложно и предложение `else` опущено, управление передается на выполнение оператора, следующего за оператором `if`.

Пример:

```
if (i>0)   y=x/i;
else
  {
    x=i;
    y=f(x);
  }
```

В примере выполняется оператор `y=x/i`, если `i` больше нуля. Если `i` меньше или равно нулю, то значение `i` присваивается переменной `x`, а значение функции `f(x)` присваивается переменной `y`.

Синтаксис оператора switch:

```
switch (<выражение>) {
[case <константное выражение>:<оператор>]
...
[default: <оператор>]
}
```

Оператор switch передает управление тому оператору своего тела, для которого case-константное выражение равно значению switch-выражения в круглых скобках.

Выполнение тела оператора начинается с выбранного оператора и продолжается до конца тела или до тех пор, пока очередной оператор не передает управление за пределы тела.

Оператор default выполнится, если ни одно из case-константных выражений не равно значению switch-выражения. Если соответствующее case-константное выражение не найдено, а default-оператор опущен, то оператор switch не выполняется.

Switch-выражение — это целое размера int или короче. Если <выражение> короче, чем int, оно расширяется до int.

Каждое case-константное выражение преобразуется к типу switch-выражения. Значение каждого case-константного выражения должно быть уникальным внутри тела оператора.

Метки case и default в теле оператора switch существенны только при начальной проверке, когда определяется стартовая точка для выполнения тела оператора. Все операторы, появляющиеся между стартовым оператором и концом тела, выполняются независимо от их меток.

Пример:

```
switch (c)
{
  case 'A':
    ULetter++;
  case 'a':
    LLetter++;
  default:
    total++;
}
```

Все три оператора в теле switch выполняются, если c = 'A'. Передача управления осуществляется на первый оператор

ULetter++, далее операторы выполняются в порядке их следования в теле.

Если c равно 'a', переменные LLetter и total инкрементируются. Наконец, если c не равно ни 'A', ни 'a', то инкрементируется только переменная total.

<Оператор> в теле switch может быть составным, т. е. иметь следующий синтаксис:

```
{<оператор> [; <оператор>]...
}
```

Например:

```
switch (i)
{
  case -1:
    {n++;
    break;
}
  case 0:
    {z++;
    break;
}
  case 1:
    {p++;
    break;
}
}
```

В теле switch после каждого оператора следует оператор break. Оператор break осуществляет принудительный выход из switch. Последний оператор break не является обязательным, поскольку без него управление было бы передано из тела на конец составного оператора.

В Java определена сокращенная версия условного оператора. Она имеет следующее синтаксическое представление:

```
<выражение>?<операнд 1>:<операнд 2>
```

<Выражение> вычисляется с точки зрения его истинности. Если оно истинно (имеет ненулевое значение), то результатом условного оператора является значение выражения <операнд 1>. Если <Выражение> равно нулю — вычисляется <операнд 2> и результатом является значение выражения <операнд 2>.

Пример:

```
j = (i<0) ? (-i) : (i);
```

В примере j присваивается абсолютное значение i. Если i меньше нуля, то j присваивается -i. Если i больше или равно нулю, то j присваивается i.

Операторы цикла. Язык Java включает следующие три оператора цикла: while, do-while, for.

Синтаксис оператора while:

```
while (<выражение>)
  <оператор>
```

Вначале вычисляется <выражение>. Если <выражение> ложно, то тело оператора while не выполняется, и управление передается следующему оператору программы. Если <выражение> является истиной (отлично от нуля), то выполняется тело оператора. Перед каждым следующим выполнением тела оператора <выражение> перевычисляется. Оператор while может завершиться при выполнении операторов break, goto, return внутри тела while.

Пример:

```
while (i>=0)
{
  intArray1[i] = intArray2[i];
  i--;
}
```

В примере копируются значения из intArray2 в intArray1. Если i больше или равно нулю, то происходит копирование i-го значения и i декрементируется. Когда i становится меньше нуля, то выполнение оператора while завершается.

Синтаксис оператора do-while:

```
do
<оператор>
while (<выражение>);
```

Тело оператора do выполняется один или несколько раз (цикл с постусловием), пока <выражение> истинно. Вначале выполняется <оператор> тела, затем вычисляется <выражение>. Если результат — ложь, то оператор do завершается и управление передается следующему оператору в программе. Если выражение истинно (не равно нулю), то тело оператора выполняется снова и снова проверяется <выражение>. Выполнение тела опе-

ратора продолжается до тех пор, пока <выражение> не станет ложным. Оператор do может также завершить выполнение при выполнении операторов break, goto или return внутри тела оператора do.

Пример:

```
do {
  y=f(x);
  x--;
} while (x>0);
```

Вначале выполняются два оператора y=f(x); и x--; независимо от начального значения x. Затем вычисляется выражение x>0. Если x>0, то тело оператора выполняется снова, и выражение x>0 перевычисляется. Тело оператора выполняется до тех пор, пока x не станет меньше или равным нулю.

Синтаксис оператора for:

```
for ([<инициализация>];[<условие>];[<выражение цикла>])
      <оператор>
```

Тело оператора for (цикл с предусловием) выполняется ноль и более раз до тех пор, пока <условие> не станет ложным. Выражения инициализации и цикла могут быть использованы для инициализации и модификации величин во время выполнения оператора for.

Первым шагом при выполнении оператора for является вычисление выражения инициализации, если оно имеется. Затем вычисляется условное выражение:

1. Если условное выражение истинно (не равно нулю), выполняется тело оператора. Затем вычисляется выражение цикла (если оно есть). Процесс повторяется снова с вычисления условного выражения.

2. Если условное выражение опущено, его значение принимается за истину, и процесс выполнения продолжается в соответствии с п. 1. В этом случае оператор for может завершиться только при выполнении в теле оператора операторов break, goto, return.

3. Если условное выражение ложно, выполнение оператора for заканчивается и управление передается следующему оператору программы.

Пример:

```
for (i=space=tab=0; i<MAX; i++)
{
  if (line[i] == '\x20')
      space++;
    if (line[i] == '\t') {
      tab++;
      line[i] = '\x20';
  }
}
```

В приведенном примере подсчитывается количество символов «пробел» ('\x20') и «табуляция» ('\t') в массиве символов, поименованном line, и замена каждого символа табуляции на пробел. Сначала i, space и tab инициализируются нулем. Затем i сравнивается с константой MAX. Если i меньше MAX, то выполняется тело оператора. В зависимости от значения line[i] выполняются операторы if. Затем переменная i инкрементируется и снова сравнивается с константой MAX. Тело оператора выполняется до тех пор, пока значение i не станет больше или равно MAX.

Если какие-либо переменные цикла при инициализации объявляются, необходимо помнить, что областью видимости этих переменных является только тело цикла. Следовательно, приведенный ниже пример вызовет ошибку:

```
int j = 10;
for (int i=0; i < 20; i++) {
   j++;
}
j += i ; // i вне области видимости
```

Операторы перехода. В Java нет оператора goto, поэтому управление на метки передается с помощью операторов перехода. Язык Java имеет следующие операторы перехода: break, continue, return.

Синтаксис оператора break:

```
break [<Идентификатор>];
```

Оператор break прерывает выполнение цикла (for, do, while), завершает оператор switch или выполняет переход по указанной метке на более высокий уровень вложенности.

Синтаксис оператора continue:

```
continue [<Идентификатор>];
```

Используется в конструкциях `while`, `for` и `switch`. Прерывает выполнение текущего прохода цикла и передает управление на начало следующего прохода. Указание метки в операторе `continue` позволяет выбрать уровень вложенности цикла, которому передается управление.

Синтаксис оператора `return`:

```
return <Выражение>;
```

Оператор `return` служит для передачи возвращаемого значения выражения из метода, конструктора или блока инициализации статической переменной.

Операторы исключений. В языке Java аппарат обработки исключительных ситуаций реализован средствами операторов `try`, `throw`, `catch` и `finally`.

Оператор `try` определяет блок, в котором возможен перехват исключений.

Оператор `throw` прерывает для указанного исключения выполнение программы. Управление передается соответствующему обработчику исключений.

Оператор `catch` определяет перехватываемое исключение и блок обработки этого исключения. Для одного блока `try` может быть указано несколько `catch`-блоков обработки исключений.

Оператор `finally` задает блок, обязательно выполняемый при завершении. В этот блок, как правило, включаются операторы освобождения ресурсов, закрытия потоков и файлов.

Пример:

```
try{
if (i==0) { throw new Exception1;} // Прерывание в программе
                                   // и инициализация исключения
} catch (Exception1 er)
{ System.out.println("Перехвачено исключение Exeption1"); }
finally { System.out.println ("Выполняется завершающий блок");}
```

Любое исключение — это объект, являющийся экземпляром класса `Exeption` (в примере — `er`). Для определения своего собственного исключения необходимо объявить класс для этого исключения.

Блок. *Блоком* называется последовательность объявлений переменных и операторов, заключенная в фигурные скобки.

Оператор определения типа объекта. Оператор `instanceof` устанавливает принадлежность объекта указанному классу или интерфейсу, например:

```
if (Obj instanceof Button) {
// Объект Obj может быть приведен к типу Button.
....
  }
```

Оператор import. Этот оператор позволяет указать имена пакетов и классов, используемых при компиляции.

Оператор `import` может быть записан следующим образом:

```
import PackageName;              // Имя пакета
import PackageName.Identifier;  // Имя пакета и имя класса
import PackageName.*;  // Имя пакета и все входящие в него
                       // классы
```

Использование оператора `import` позволяет обращаться к доступным компонентам указанного пакета без квалификации их именем пакета.

Структура программы

Язык Java позволяет создавать как самостоятельные приложения, так и апплеты (программы, которые выполняются под управлением web-браузеров). Любая программа Java может описывать произвольное количество классов, которые при компиляции записываются в отдельные файлы, т. е. один файл программы с расширением .java может породить после компиляции несколько файлов байт-кода с расширением .class.

Программа-приложение всегда должна содержать метод `main`, называемый главным методом. Этот метод позволяет принимать аргументы командной строки и является первым выполняемым методом.

Рассмотрим пример простого приложения.

```
import java.lang.* ; // необязательное подключение библиотеки
class HelloJava {
   public static void main (String args [])
{                              // Обязательный метод main
       System.out.println("Hello, Java");
}
}
```

В приведенном примере оператор `import` позволяет использовать все методы пакета java.lang. Метод `System.out.println` выводит на экран строку текста, указанную в качестве аргумента. Программа содержит один класс `HelloJava`, который имеет только главный метод `main`. Метод `main` в качестве параметра обязательно имеет объект `args[]`, представляющий собой массив строк — аргументов командной строки вызова приложения.

Исходный файл приложения Java может содержать только один класс `public`, причем имя файла должно в точности совпадать с именем такого класса. В данном случае исходный файл называется HelloJava.java (если бы файл назывался, например, hellojava.java, компилятор выдал бы сообщение об ошибке).

JDK (Java Developer's Kit — инструментальный набор разработчика Java) содержит иерархию классов, интерфейсов и абстрактных классов. Собственно говоря, все возможности программирования на Java базируются на классах этого инструментального набора. JDK подразделяется на несколько отдельных пакетов, называемых библиотеками:

```
java.applet  java.awt  java.awt.image  java.awt.peer     java.net
java.io      java.lang java.util       java.tools.debug
```

Объявление переменных

Синтаксис объявления переменной следующий:

```
[<область видимости>] [<модификатор>] <тип данных>
        <имя переменной> [= <инициализация>]
```

Для переменных (свойств) класса *область видимости* может иметь следующие (взаимоисключающие) значения:

- `public` — разрешает наследование свойства и доступ к нему как в текущем пакете, так и из других пакетов;
- `protected` — разрешает доступ к свойству только в рамках текущего пакета, а наследование — и для текущего, и для других пакетов;
- `private` — разрешает доступ к свойству только в рамках класса, в котором оно объявлено.

Если область видимости не указана, по умолчанию разрешается наследование и доступ к свойству в рамках одного (текущего) пакета.

Модификатор переменной указывает, является ли свойство константным:

- `final` — запрещает присвоение значений переменной вне тела класса, в котором она объявлена. Переменная, имеющая модификатор `final`, всегда должна быть инициализирована внутри тела класса и может рассматриваться как константа класса;
- `static` — означает, что переменная является переменной класса (*статической*), а не переменной экземпляра класса. Каждый экземпляр класса при создании получает свою «независимую копию» всех нестатических свойств и методов, поэтому нестатические свойства и методы в разных экземплярах класса могут иметь разные значения. Статические же свойства и методы принадлежат не экземплярам класса, а самому классу и будут одинаковы для всех экземпляров класса. При этом изменение статического свойства в любом экземпляре класса приведет к его синхронному изменению для остальных экземпляров класса.

Инициализация переменных. Переменная может быть проинициализирована одновременно с объявлением, т. е. ей может быть присвоено некоторое значение (или может быть выполнено размещение переменной с помощью оператора `new`).

Инициализация переменной, объявляемой с модификатором `static`, выполняется сразу при загрузке класса. Инициализация переменной экземпляра класса выполняется при запуске метода-конструктора класса. Инициализация локальной переменной осуществляется при выполнении оператора объявления этой переменной.

Примеры объявления переменных:

```
int X, Y;          // Объявление переменных целого типа
float f = 1.0;     // Объявление переменной вещественного типа
java.lang.String String1 = "новая строка";
                   // Объявление объекта строкового типа
Exception er = new Exception ();
                   // Объявление объекта-исключения
double dArray [] = new double [100];
                   // Объявление объекта-массива
```

Следует помнить, что при инициализации переменных экземпляра класса нельзя допускать ссылок на еще не инициали-

зированные переменные. Например, следующий фрагмент кода вызовет ошибку во время компиляции:

```
class cClassA   {int i = j + 1; int j = 1;}
```

Инициализация статических переменных. *Статические переменные* ассоциируются не с объектом, а с классом. Как уже отмечалось выше, для каждого класса создается только один экземпляр статической переменной (переменной класса).

При описании статических переменных необходимо помнить, что их объявление и инициализация должны быть выполнены вне всякого метода.

Статические переменные не могут содержать ссылки на переменные экземпляра своего класса.

При инициализации статических переменных можно вызывать статические методы.

Инициализация статических переменных выполняется при загрузке соответствующего класса в порядке, описанном в объявлении класса.

Пример:

```
class ClassA {
static int a = 1;
static double b;
static { a++; c = 100; }
static int c = 1;       // Предыдущее значение переменной c
                        // теряется
static Window d = new Window ( );
}
```

Определение метода

Методом называется программный код (подпрограмма), вызов которого можно выполнить как внутри класса, так и для объекта класса. Метод может иметь модификаторы доступа, возвращать значение и получать параметры.

Синтаксис объявления метода следующий:

```
[<модификатор_доступа>] [<спецификаторы>]
    <тип_возвращаемого_значения> <имя__метода> ([<параметры>])
[throws <список_исключений_которые_возвращает_класс>]
{ /* операторы */
[return <возвращаемое_значение>] ;}
```

Модификатор доступа может отсутствовать или иметь одно из следующих значений: `public`, `protected` или `private`.

Смысл наличия какого-либо значения модификатора доступа (а также и его отсутствия) аналогичен значению этих модификаторов для переменных и был подробно рассмотрен в разделе объявления переменных.

Спецификатор `static`. Задает *метод класса* (или *статический метод*). Метод может быть вызван даже в том случае, если не создано ни одного экземпляра класса, содержащего отмеченный метод. Этот метод всегда вызывается непосредственно из класса. Метод класса имеет доступ ко всем переменным и методам этого класса, объявленным со спецификатором `static`. Метод, не имеющий спецификатора `static`, является *методом экземпляра*. Метод экземпляра может быть вызван только для созданного экземпляра класса или подкласса. Такой метод нельзя вызвать непосредственно из класса.

Отметим, что синтаксис языка Java не требует, чтобы совместно со спецификатором `static` указывался спецификатор `final`, но неявно это подразумевает. Поэтому любая попытка *переопределения* метода класса (о переопределении методов см. ниже) будет ошибочной.

Спецификатор `abstract`. Абстрактные методы объявляются, но не реализуются в данном классе. Тело метода должно быть описано в подклассах текущего класса. Ни `static`-методы, ни конструкторы классов не могут объявляться как `abstract`. Более того, абстрактные методы нельзя определять как `final`, поскольку в этом случае их нельзя будет переопределить.

Иногда требуется создать только некоторую оболочку метода и не раскрывать на данном уровне его реализацию. Впоследствии реализацию можно конкретизировать, например, с учетом различных аппаратных платформ. Такие методы отмечаются спецификатором `abstract` и называются *абстрактными методами*. Абстрактный метод никогда не имеет тела метода (вместо фигурных скобок, ограничивающих тело метода, объявление метода завершается точкой с запятой). Абстрактные методы можно объявлять только в абстрактных классах. Объявление абстрактного метода в классе, не имеющем модификатора `abstract`, приведет к ошибке компиляции. Любой подкласс абстрактного класса, являющийся сам, в свою очередь, также абстрактным классом, обязательно должен описывать реализацию всех ранее не реализованных абстрактных методов суперкласса.

Отметим, что методы, объявленные с модификатором `private`, не могут быть абстрактными, так как они недоступны вне тела класса и, следовательно, не могут быть реализованы.

Методы класса также не могут выступать в качестве абстрактных методов, так как считаются конечными и не могут быть переопределены.

Спецификатор `final`. Любые подклассы текущего класса не смогут переопределить такой метод. Эта возможность увеличивает защищенность классов и гарантирует, что операции, определенные в данном методе, нельзя будет изменить.

Объявление метода со спецификатором `final` предотвращает его дальнейшее переопределение. Такие методы иногда называются *конечными методами*.

По умолчанию подразумевается, что локальные (`private`) методы являются также и конечными методами.

Спецификатор `native`. Объявляет методы, написанные на других языках программирования. Обычно такие методы пишутся на C++ для ускорения работы критических участков программы. Для определения этих методов нужно поместить спецификатор `native` в начале объявления метода и вместо тела метода поставить точку с запятой, например: `native void toggleStatus();`

`Native`-методы могут наследоваться, иметь спецификаторы `static` или `final`, переопределяться и во всем остальном ведут себя как обычные методы языка Java.

Спецификатор `synchronized`. Позволяет защитить данные, которые могут быть разрушены в том случае, если два метода из разных потоков пытаются одновременно обратиться к одним и тем же данным. `Syncronized`-метод не может начать работать со свойствами, пока их не освободит другой метод.

`Synchronized`-методы перед выполнением будут использовать управляемую блокировку: для метода класса выполняется блокировка класса, а для метода экземпляра — блокировка объекта.

Тип возвращаемого значения. Если метод возвращает какое-либо значение, то при его объявлении следует указать тип возвращаемого значения, а тело метода должно содержать оператор `return`. В противном случае метод должен быть объявлен с ключевым словом `void`, указывающим на отсутствие возвращаемого значения.

Если в качестве возвращаемого значения должен быть использован массив, то после имени метода может быть указана пара квадратных скобок.

Список параметров. Список формальных параметров содержит упорядоченные описания параметров, разделенные запятой. Описание параметра состоит из типа параметра и имени, используемого для идентификации этого параметра внутри метода. Если в качестве типа параметра используется массив, после имени параметра может быть указана пара квадратных скобок (в качестве составной части указания типа).

Если метод не имеет параметров, передаваемых ему при вызове, после имени метода указывается пара круглых скобок.

Параметры рассматриваются как локальные переменные метода, специфицированные вне тела метода. При вызове метода перед его выполнением происходит вычисление выражений, представляющих передаваемые методу фактические параметры, и присвоение вычисленных значений соответствующим локальным переменным (формальным параметрам).

Методы, обрабатывающие исключения. Java позволяет обрабатывать исключения (именованные ошибки) внутри метода или передавать их далее в ту часть программы, которая вызвала этот метод. Если исключение обрабатывается внутри метода, обработчик указывается ключевым словом `catch`. В противном случае (обработка исключения вне метода) следует при объявлении метода указать ключевое слово `throws` и список «бросаемых» из метода исключений.

Выбор места обработки исключений осуществляется по иерархическому принципу:

- если исключение не указано в списке после ключевого слова `throws`, обработка должна быть определена и выполнена внутри метода;
- если исключение указано в списке после ключевого слова `throws`, оно обрабатывается:

  1) в коде метода, вызвавшем данный метод, в том случае, если этот вызвавший метод не содержит указанное исключение в своем списке после ключевого слова `throws` (при этом вызвавший метод обязательно должен иметь обработчик исключения, определенный оператором `catch`);

  2) в коде метода далее по иерархии вызовов, предусматривающем обработку этого исключения.

***Использование* `this` *и* `super`.** При вызове метода экземпляра ключевое слово `this` означает ссылку на текущий объект. Оно может быть использовано для передачи самого объекта (экземпляра класса) в качестве аргумента методу этого экземпляра класса.

Пример:

```
class ClassA extends classB {
void Method1 (ClassC oObject) {
oObject.Method2(this) ;  // Вызов метода из класса ClassC
          // с передачей в качестве параметра текущего объекта
}
}
```

Ключевые слова `this` и `super` используются также и для указания квалифицированной ссылки на поля (переменные и методы) экземпляра класса или суперкласса.

По умолчанию любой метод первоначально всегда ссылается на свои собственные переменные и методы, и только в случае их отсутствия выполняется поиск этих полей по иерархии суперклассов, например:

```
class ClassA { int a;
...
}
class ClassB extends ClassA {
int a, b, c;
void Metod1 ( ) {
println (a + " \n");      // Ссылка а эквивалентна ссылке
                          // this.a
println (super.a);        // Ссылка на а из класса ClassA
println (ClassA.a);       // Ссылка на а из класса ClassA
...
}
```

Имя суперкласса также может быть использовано для квалифицированного доступа к переменным или методам экземпляра класса:

```
class ClassA {
       Object x;
 }
class ClassB extends ClassA {
       float x;
 }
class ClassC extends ClassB {
       char x;
```

```
        void Metod1 ( ) {
          char cX = x;  // Присваивается значение переменной x
                        // из класса ClassC (тип char)
          float fX = ClassB.x; // Присваивается значение
                               // переменной x из класса
                               // ClassB (тип float)
                               // ClassB.x эквивалентно super.x
          Object oX = ClassA.x;
                      // Присваивается значение переменной
                      // x из класса ClassA (тип Object)
                    }
}
```

Переопределение методов. Переопределением метода называется объявление в подклассе метода с именем и списком параметров, в точности совпадающими с ранее объявленными в суперклассе. В списке параметров должны совпадать как число параметров, так и их типы.

При переопределении метода существует правило расширения доступа: переопределенный метод не должен иметь более «узкую» область видимости. Использование модификаторов доступа должно удовлетворять следующим условиям:

- если переопределяемый метод суперкласса не содержит ни одного из модификаторов доступа `public`, `protected` или `private`, то переопределяющий метод не может иметь модификатора `private`;
- если переопределяемый метод суперкласса имеет модификатор доступа `protected`, то переопределяющий метод должен иметь модификаторы доступа `public` или `protected`;
- если переопределяемый метод суперкласса имеет модификатор доступа `public`, то переопределяющий метод может иметь только модификатор доступа `public`.

Тип возвращаемого значения переопределяющего метода должен совпадать (или преобразовываться оператором присваивания) с типом возвращаемого значения переопределяемого метода.

Если класс содержит переопределяющий метод, то переопределенный метод следует вызывать с квалификацией ссылки ключевым словом `super` или именем суперкласса, например:

```
class ClassA {
public Metod1() {
...
}}
```

```
class ClassB extends ClassA {
public Metod1() {   ... }
...
Metod1()          // Вызов метода из класса ClassB
super.Metod1()    // Вызов метода из класса ClassA
...
}
```

Отметим, что метод `private` не является доступным для подклассов и поэтому не может быть переопределен.

Определение метода-конструктора. *Конструктором* называется специальный метод, используемый для инициализации создаваемых объектов (экземпляров класса).

Синтаксис определения конструктора отличается от синтаксиса определения метода следующим:

- отсутствуют спецификаторы метода;
- отсутствует оператор `return`.

Имя конструктора всегда должно совпадать с именем класса, содержащего объявление конструктора.

Конструктор не может использовать оператор `return` для возврата значения. По умолчанию тип возвращаемого значения для конструктора всегда `void`.

Класс может не иметь своего конструктора. В этом случае при создании объекта используется конструктор по умолчанию. Такой конструктор получает параметры и вызывает непосредственно конструктор суперкласса `super()`.

В отличие от методов конструкторы не наследуются подклассами.

Тело конструктора так же, как и тело метода, заключается в фигурные скобки. Однако необходимо соблюдать следующее правило упорядочивания операторов внутри тела конструктора: если используются операторы вызова конструкторов суперкласса, то они должны быть указаны вначале, до всех остальных операторов тела конструктора.

Синтаксис тела конструктора следующий:

```
{<ВызовКонструкторов>
  <БлокОператоров>}
```

Вызов конструкторов выполняется следующими операторами:

```
this([<Список параметров>]);
super([<Список параметров>]);
```

Первый оператор конструктора может вызывать:

- другой конструктор в этом же классе, используя ключевое слово `this` с соответствующим списком параметров;
- конструктор непосредственного суперкласса, используя ключевое слово `super` с соответствующим списком параметров.

Если оператор `super` не будет указан, то любой конструктор первоначально будет выполнять неявный вызов конструктора суперкласса с пустым списком параметров. Если суперкласс не содержит конструктора с пустым списком параметров, то все объявленные переменные, не имеющие значений для инициализации, инициализируются значениями по умолчанию: числовые переменные — значением ноль, а строковые — значением `null` (пустая строка).

Обобщая рассмотренные выше правила работы конструкторов, можно заключить следующее:

- при создании объекта любого заданного класса будет неявно выполнена цепочка вызовов всех конструкторов его суперклассов;
- первым будет выполнено тело конструктора для `Object`;
- каждый последующий конструктор в цепочке будет выполняться только после выполнения конструктора своего непосредственного суперкласса;
- при создании объекта будут проинициализированы все переменные экземпляра.

Пример:

```
class CPoint  {
   double x, y;
   Color C = yellow;
   Point new(float fXValue, float fYValue) {
   // В этом месте выполняется неявный вызов
   // конструктора суперкласса super()
   // и происходит инициализация переменной C значением yellow
    x = fXValue; y = fYValue;
   }
Point new( )   {    // Конструктор по умолчанию
this(1.0, 1.0);     // Значения по умолчанию
}
```

Классы и интерфейсы

Иерархия классов и интерфейсов. *Объявление класса* создает новый класс, который является производным от другого класса,

называемого *непосредственным суперклассом*. Класс расширяет (`extends`) свой непосредственный суперкласс своими методами (с их реализацией) и переменными (свойствами). Такая иерархия реализации классов Java обеспечивает повторное использование программного кода.

Каждый объект является *экземпляром* (`instance`) некоторого класса.

Класс `A` является суперклассом класса `C` в том случае, если выполнено одно из следующих условий:

- `A` — это сам `C`;
- `A` есть непосредственный суперкласс для `C`;
- есть такой класс `B`, что `A` является суперклассом для `B` и `B` является суперклассом для `C`.

Переменные, объявленные переменными класса `C`, могут иметь в качестве своего значения ссылку на объект, который является экземпляром класса `C` или экземпляром любого его подкласса.

Объявление интерфейса создает новый ссылочный тип данных, который специфицирует описания методов и имена некоторых констант, но не определяет саму реализацию. Интерфейс может быть объявлен с целью непосредственного расширения одного или нескольких интерфейсов.

Интерфейс `K` является расширением интерфейса `I` в том случае, если выполнено хотя бы одно из следующих условий:

- `K` совпадает с `I`;
- `K` является непосредственным расширением `I`;
- существует такой интерфейс `J`, что `K` является расширением `J` и `J` является расширением `I`.

Описание класса. Класс — это описание свойств и методов объекта. После компиляции байт-код каждого класса размещается в отдельном файле. Синтаксис описания класса следующий:

```
[<модификатор_доступа>] [<спецификатор>] class <имя_класса>
                        [extends <имя_класса_родителя>]
                        [implements <список_имен_интерфейсов>]
                        {<тело класса>}
```

Модификатор доступа. Если модификатор доступа имеет значение `public`, класс доступен для всех объектов. Класс `public` должен обязательно содержаться в файле, имеющем такое же название, что и имя класса. Если модификатор доступа отсутствует, это означает, что класс доступен только для объектов, находящихся в том же пакете.

Спецификатор. Спецификатор класса может принимать одно из двух следующих значений:

`final` — указывает на то, что класс не может иметь подклассов и, соответственно, не должен появляться в предложении `extends` объявления другого класса;

`abstract` — указывает на то, что класс является абстрактным, т. е. может иметь абстрактные методы.

Задание одновременно при объявлении класса модификаторов `final` и `abstract` вызывает ошибку компиляции, так как предполагается, что любой абстрактный класс создается с целью последующей разработки некоторых подклассов, реализующих абстрактные методы этого класса.

Класс-родитель. Любой класс может иметь только одного родителя (или суперкласс): множественное наследование в языке Java не поддерживается. Суперкласс указывается ключевым словом `extends`. Используемый суперкласс должен быть доступным и не иметь модификатора `final`.

По умолчанию предполагается, что если для класса не указано никакого суперкласса, его непосредственным родителем будет класс `Object`. Класс `Object` является корневым классом в формируемой иерархии классов, таким образом всегда является суперклассом любого другого класса.

Интерфейсы. Объявление класса может содержать список интерфейсов. Интерфейсы задаются ключевым словом `implements`. Если класс, для которого указываются интерфейсы, не является абстрактным классом, то все методы, объявленные в этих интерфейсах, должны быть определены или в самом этом классе, или в некотором его суперклассе.

Тело класса. Тело класса представляет собой блок (заключается в фигурные скобки) и состоит из списка объявлений полей класса. Этот список может не содержать ни одного элемента. Полем класса могут являться переменная или метод. Метод, имя которого совпадает с именем класса, называется *конструктором* или *методом-конструктором.*

Примеры:

```
public final class MyBook extends Book implements Reader {
                    // Свойства и методы класса
}
class MyClass   {
                    // Свойства и методы класса
    }
```

Приведение классов. В языке Java имя класса является именем типа (ссылочный тип), поэтому может быть указано в качестве типа переменной. Все переменные, имеющие ссылочный тип, указываемый именем класса, являются объектами.

Тип объекта может быть определен с помощью оператора `instanceof`, например:

```
// Определение типа объекта ObjectName
String Str1;
MyObj Str2;
Object ObjectName;
if (i=1) {
     ObjectName = (Object) Str1;}
else {
     ObjectName = (Object) Str2;}
if (ObjectName instanceof String) {
     String Str1_dub = (String) ObjectName;
         // Код для объекта типа String
}
if (ObjectName instanceof MyObj) {
     MyObj Str2_dub = (MyObj) ObjectName;
         // Код для объекта типа MyObj
}
```

В данном примере объект сначала был приведен к своему косвенному суперклассу `Object`, а затем определен непосредственный тип объекта.

При приведении типов справедливы следующие правила:

- объект всегда может быть приведен к типу своего непосредственного суперкласса;
- приведение ссылочных типов может выполняться по иерархии классов как угодно глубоко;
- любой класс ссылочного типа всегда можно привести к типу `Object`.

Следует отметить, что после приведения объекта к типу суперкласса все переменные и методы самого класса объекта становятся недоступными для приведенного объекта.

Объявление интерфейса. *Интерфейсом* называется набор объявлений констант и методов без описания их реализации. Реализация интерфейса выполняется в классе, который использует данный интерфейс.

В языке Java отсутствует множественное наследование, т. е. любой объект может порождаться только от одного объекта. Но иногда необходимо, чтобы объект унаследовал не только свойст-

ва и методы родителя, но и свойства и методы другого объекта. Для этих целей используются интерфейсы — классы, в которых объявлены, но не описаны методы. Класс, реализующий интерфейс, должен сам описать реализацию методов интерфейса. При описании методов интерфейсов можно не указывать модификаторы доступа, все они по умолчанию рассматриваются как `public`. Другие модификаторы доступа в описании методов интерфейса использовать нельзя. В интерфейсах также нельзя использовать переменные, не являющиеся константами (`final`). Любые поля интерфейса, вне зависимости от того, указаны модификаторы доступа или нет, рассматриваются как `public`, `final` и `static`.

Синтаксис объявления интерфейса следующий:

```
[<модификатор>] interface <имя_интерфейса>
                                    [extends <список_интерфейсов>]
{   // Объявление свойств, объявление методов без описания
    // их реализации
}
```

Пример:

```
public interface Radio {
final String nazvanie = "Автомобильный радиоприемник";
                              // Описание константы
public long Volna();          // Объявление метода
    }
```

Интерфейс применяется к объекту при помощи ключевого слова `implements`, например:

```
class mercedes extends avtomobil implements Radio
// К классу mercedes применяется интерфейс Radio.
```

Модификаторы интерфейса. Интерфейс может иметь модификаторы `public` и `abstract`.

Если интерфейс объявлен с модификатором доступа `public`, то к нему может быть произведен доступ из других пакетов. В противном случае интерфейс может быть использован только в том пакете, в котором он объявлен. Один модуль компиляции может содержать только один интерфейс с модификатором `public`. Одновременно в одном модуле компиляции может быть один интерфейс и один класс с модификаторами `public`.

Каждый интерфейс по умолчанию имеет модификатор `abstract` (который не обязательно указывать при объявлении).

Тело интерфейса. Тело интерфейса заключается в фигурные скобки аналогично телу абстрактного класса. Но тело интерфейса не может содержать конструктора или блоков инициализации статических переменных.

Объявление переменных интерфейса. Каждая переменная интерфейса по умолчанию считается переменной с модификаторами `public`, `static` и `final`. Каждая переменная в теле интерфейса обязательно должна быть инициализирована (инициализирующее выражение должно быть константой).

Объявление методов интерфейса. Каждый метод, объявленный в теле интерфейса, по умолчанию считается методом с модификаторами `public` и `abstract`. Объявление метода завершается точкой с запятой и не содержит тела метода, заключенного в фигурные скобки.

Метод интерфейса не может иметь модификаторов `final` или `static`.

Создание объекта. При создании объекта последовательно выполняются следующие действия:

- создается новый объект указанного типа, и все его переменные инициализируются своими значениями по умолчанию;
- для создания объекта выполняется соответствующий конструктор с указанным списком параметров. При использовании метода `newInstance` вызывается конструктор без списка параметров;
- после выполнения конструктора формируется ссылка на созданный и инициализированный объект, которая и является значением выражения, создающего объект.

Объект может быть создан вызовом метода `newInstance` из класса `Object`. При таком способе создания объекта последовательно выполняются следующие действия:

- при создании объекта используется вызов конструктора без списка параметров;
- тип создаваемого объекта соответствует типу класса, для которого выполняется метод `newInstance`. Все переменные экземпляра созданного объекта инициализируются своими значениями по умолчанию;
- возвращаемым значением метода `newInstance` является ссылка на созданный и инициализированный объект.

Массивы

Массивы традиционно представляют собой набор однотипных элементов с адресным доступом к каждому отдельному элементу. Кроме этого, массивы в языке Java являются динамически размещаемыми объектами и для них можно применять все методы класса `Object`.

Java поддерживает работу с одномерными массивами. Нумерация элементов массива начинается с нуля. Хотя язык Java и не поддерживает непосредственно работу с многомерными массивами, в качестве элементов массива могут быть использованы объекты, являющиеся массивами. В свою очередь, элементы подмассива также могут быть массивами и т. д.

Существует два пути размещения массива: с помощью оператора `new` или средствами инициализации массива, использующими оператор равенства (=) в объявлении переменной массива.

Объявление массива. Синтаксис объявления массива, в основном, совпадает с синтаксисом объявления переменной. Различие состоит в том, что тип данных определяет тип элементов массива (и может быть любым допустимым типом языка Java), а после типа указываются открывающая и закрывающая квадратные скобки ([]). Квадратные скобки могут появляться до и/или после имени массива. Число пар квадратных скобок определяет уровень вложенности массива.

Объявление массива создает переменную, которая будет содержать ссылку на массив. Для размещения массива в памяти одновременно с объявлением переменную массива следует инициализировать: использовать оператор `new` для создания ссылки на объект-массив или указать список инициализации, заключенный в фигурные скобки. При использовании для создания массива списка значений, заключенного в фигурные скобки, длина массива определяется числом заданных элементов. Элемент массива может быть задан выражением, тип которого должен соответствовать типу массива.

Примеры:

```
// Следующие массивы будут только объявлены,
// но не размещены в памяти
int [ ]cInteger;    // Массив элементов типа integer
float [ ][ ]cfloat;// Массив массивов элементов типа float
Object [ ]cObject; // Массив переменных типа object
// Следующие массивы будут и объявлены,
// и одновременно размещены в памяти
```

```
int [ ]cInteger = new int[10];
int [ ]iAr= {1,3,5,7,9,11,13,15,17,19};
          // Массивы переменных типа integer из 10 элементов
Exception aExc[ ] = new Exception[3];
Object  aObject[ ][ ] = new Object[2][3];
```

Каждый массив имеет длину, определяемую переменной `length`, которая имеет модификатор `final` (т. е. является константой и после размещения массива не может быть изменена). Массивы переменной длины непосредственно не могут быть созданы, однако такие массивы можно создавать путем размещения другого массива требуемой длины и присвоения переменной массива ссылки на новый массив.

В качестве индексов массива могут использоваться значения типа `int`, а также значения типа `short`, `byte` или `char` (так как они могут быть преобразованы к значению типа `int`). Значения типа `long` не могут использоваться в качестве индексов массива.

Доступ к элементам массива выполняется по следующей схеме:

```
<Имя массива> [<Выражение>] ... [<Выражение>]
```

Массивы строк и символов. Язык Java позволяет создавать массивы строк и массивы символов. Массив символов при этом строкой не является, и для него нельзя использовать методы обработки строк. Ни строка, ни массив символов в таком случае не должны прерываться символом `'\u0000'` (символ `NUL`).

Примеры:

```
char cChar[ ] = {'n','o',t,' ','a',' ','S','t', 'r',
'i','n','g'};
String []cString = {"array", "of", "String"};
```

Классы, определяющие работу со строками

Работу с привычными переменными строкового типа в языке Java обеспечивают два стандартных класса — `String` и `StringBuffer`. Эти классы объединяют данные — массивы символов (строки) и наиболее распространенные методы, необходимые для создания, анализа и манипулирования такими массивами. Класс `String` используется для представления строк-констант, а класс `StringBuffer` — для динамических строк.

***Класс* `String`.** Класс предназначен для создания и использования константных строк. Нумерация символов в строке — от 0 (первый символ) до `length-1` (последний символ).

Объект типа `String` автоматически создается для любой взятой в двойные кавычки последовательности символов, например:

```
"Это - строка"  // Создается объект String
```

Такая способность языка делает доступными все методы класса, т. е. правомерно следующее использование:

```
int len = "Это - строка".length();
     // Возвращает число символов в строке
```

Некоторые методы класса представлены в табл. 2.5.

Таблица 2.5. **Методы класса `String`**

Метод	Описание
`String()`	Конструктор пустой строки (`Null String`)
`String(String value)`	Конструктор с аргументом `value` типа `String`
`String(char value[])`	Конструктор с аргументом `value` типа символьный массив
`String(char value[], int offset, int count)`	Конструктор с аргументом `value` типа символьный массив: `offset` — смещение в массиве символов, `count` — длина значения объекта `String`
`String(byte ascii[], int hibyte)`	Конструктор с аргументом `ascii` типа массив байтов: `hibyte` — значение старшего байта Unicode, вставляемое в старший байт каждого символа строки
`String(byte ascii[], int hibyte, int offset, int count)`	Конструктор с аргументом `ascii` типа массив байтов: `hibyte` — значение старшего байта Unicode, вставляемое в старший байт каждого символа строки; `offset` — смещение в массиве байтов; `count` — длина значения объекта `String`
`int length()`	Возвращает длину значения объекта (строки)
`char charAt(int index)`	Возвращает символ со смещением `index`
`void getChars(int sBegin, int sEnd, char dst[], int dstBegin)`	Копирует символы из объекта в символьный массив `dst`: `sBegin` — смещение начала копирования; `sEnd` — смещение конца копирования; `dstBegin` — смещение начала копирования в `dst`

Продолжение табл. 2.5

Метод	Описание
`void getBytes(int sBegin, int sEnd, byte dst[], int dstBegin)`	Копирует символы из объекта в массив байтов `dst`: `sBegin` — смещение начала области копирования; `sEnd` — смещение конца области копирования; `dstBegin` — смещение начала области копирования в `dst`
`boolean equals(Object anObject)`	Возвращает значение `true`, если параметр `anObject` не пуст, является объектом типа `String` и содержит ту же последовательность символов, что и данная строка
`boolean equalsIgnoreCase (String anString)`	Возвращает значение `true`, если строка равна строке `anString` после преобразования прописных букв в строчные
`int compareTo (String anString)`	Сравнивает лексикографически строку с `anString`. Возвращает значение: 0, если строки равны; <0, если строка меньше `anString`; >0, если строка больше `anString`
`boolean regionMatches (int toffset, String other, into ooffset, int len)`	Сравнивает область строки с областью строки `other`: `toffset` — начало сравнения (в строке); `ooffset` — начало сравнения (в строке `other`); `len` — длина сравниваемой части (в символах)
`boolean regionMatches (boolean ignorecCase, int toffset, String other, into ooffset, int len)`	Сравнивает область строки с областью строки `other`. Если `ignorecCase = true`, игнорируется регистр символов (строчные — прописные буквы): `toffset` — начало сравнения (в строке); `ooffset` — начало сравнения (в строке `other`); `len` — длина сравниваемой части (в символах)
`boolean startsWith (String prefix)`	Проверяет, начинается ли строка с указанного префикса `prefix`
`boolean startsWith (String prefix, int offset)`	Проверяет, начинается ли строка с указанного префикса `prefix` (с учетом смещения `offset`)
`boolean endsWith (String suffix)`	Проверяет, заканчивается ли строка указанным суффиксом `suffix`
`int indexOf(int ch)`	Возвращает смещение первого появления символа `ch` или –1, если `ch` не найдено
`int indexOf(int ch, int fromindex)`	Возвращает смещение первого появления символа `ch` (начиная с fromindex) или –1, если `ch` не найден

Продолжение табл. 2.5

Метод	Описание
`int indexOf(String str)`	Возвращает смещение первого появления строки `str` или –1, если `str` не найдена
`int indexOf(String str, int fromindex)`	Возвращает смещение первого появления строки `str` (начиная с `fromindex`) или –1, если `str` не найдена
`int lastindexOf(int ch)`	Возвращает смещение последнего появления символа `ch` или –1, если `ch` не найдена
`int lastindexOf(int ch, int fromindex)`	Возвращает смещение последнего появления символа `ch` (начиная с `fromindex`) или –1, если `ch` не найден
`int lastindexOf(String str)`	Возвращает смещение последнего появления строки `str` или –1, если `str` не найдена
`int lastindexOf(String str, int fromindex)`	Возвращает смещение последнего появления строки `str` (начиная с `fromindex`) или –1, если `str` не найдена
`String substring (int beginindex)`	Возвращает подстроку, начинающуюся с `beginindex` и продолжающуюся до конца строки
`String substring (int beginidx, int endidx)`	Возвращает подстроку, начинающуюся с `beginidx` и продолжающуюся до `endidx`
`String concat (String str)`	Осуществляет конкатенацию (сцепление) строки и `str`
`String replace (char oldChar, char newChar)`	Возвращает строку, где все появления `oldChar` заменены на `newChar`
`String toLowerCase()`	Возвращает строку, преобразованную к строчным буквам
`String toUpperCase()`	Возвращает строку, преобразованную к прописным буквам
`String trim()`	Возвращает строку, в которой удалены лидирующие и завершающие символы-разделители
`String toString()`	Возвращает саму строку
`char[] toCharArray()`	Возвращает строку как массив символов
`static String valueOf (boolean b)`	Возвращает строку, значение которой равно символьному представлению `b`

Окончание табл. 2.5

Метод	Описание
`static String valueOf (char ch)`	Возвращает строку, значение которой равно символьному представлению `ch`
`static String valueOf (int i)`	Возвращает строку, значение которой равно символьному представлению `i`
`Static String valueOf (float f)`	Возвращает строку, значение которой равно символьному представлению `f`
`Static String valueOf (double d)`	Возвращает строку, значение которой равно символьному представлению `d`

В тексте строки нельзя использовать двойные или одинарные кавычки и обратную косую черту «\». Если этими символами все-таки необходимо воспользоваться, то применяют escape-последовательности, например:

```
String str = "На разных \n строках и \" в кавычках \" "
```

Рассмотрим пример класса, расширяющего класс `String` процедурой подсчета количества заданных букв независимо от регистра:

```
public class WorkStr {
  private String WorkString = null;   // Переменная класса
                                      // типа String
  private int WorkCount;   // Переменная класса типа integer
// Метод-конструктор (str – строковый параметр)
  public WorkStr (String str) {
    WorkString = new String(str.toLowerCase());
    WorkCount = WorkString.length();
}
// Метод подсчета количества символов AChar:
// возвращает количество символов

  public int numberChar (char AChar) {
    int NChar = 0;     // Обнуление счетчика букв
    int i = 0;
    while (i >= 0) {
      i = WorkString.indexOf(AChar,i);
      if i >= 0 NChar++;  // Увеличение счетчика букв на 1
return NChar;
}
}
```

Класс StringBuffer. Класс содержит переменную типа строка, т. е. строку, значение которой можно менять. Методы класса содержат механизмы перераспределения памяти, если это необходимо. Например, если памяти, отведенной под `StringBuffer`, оказывается недостаточно для размещения нового значения строки, то метод `append()` размещает новый буфер большего размера, копирует туда старое содержимое, добавляет в конец необходимое количество символов и сообщает объекту `StringBuffer`, что его значение — новый символьный массив.

Все методы, которые имеют доступ к символьному значению типа `StringBuffer`, синхронизированы. Это означает, что перед исполнением метод блокирует объект `StringBuffer`, предотвращая одновременный доступ к ресурсам объекта. Любое действие над объектом со стороны будет отсрочено до тех пор, пока первый метод не закончит работу и не разблокирует объект. Такая защита объектов необходима в случае многопоточной обработки.

Некоторые общедоступные (`public`) методы класса представлены в табл. 2.6.

Рассмотрим пример класса, расширяющего класс `StringBuffer` процедурой замены всех вхождений одного символа в строке на другой:

```
public class EditStr {
  private StringBuffer ResStringBuffer = null;
                       // Переменная класса типа StringBuffer
  private int ResCount;       // Переменная класса типа integer
// Метод-конструктор (str – строковый параметр)
  public EditStr (String str) {
    ResStringBuffer = new StringBuffer(str);
    ResCount = ResStringBuffer.length();
}
// Метод замены символа oldChar на символ newChar
  public synchronized void replaceChar (char oldChar, char
newChar) {
    for (int i = 0; i < ResCount; i++) {
      if (ResStringBuffer.charAt(i) == oldChar) {
        ResStringBuffer.setCharAt(i, newChar);
      }
    }
  }
// Метод возвращает содержимое ResStringBuffer
public String toString() {
  return ResStringBuffer.toString();
  }
}
```

Таблица 2.6. **Методы класса StringBuffer**

Метод	Описание
`StringBuffer()`	Конструктор пустого строкового буфера с длиной по умолчанию
`StringBuffer (int len)`	Конструктор пустого строкового буфера с длиной `len`
`StringBuffer (String str)`	Конструктор строкового буфера со значением `str`
`int length()`	Возвращает значение счетчика символов
`int capacity()`	Возвращает текущий размер строкового буфера
`void ensureCapacity (int minimumCapacity)`	Увеличивает размер строкового буфера, если его текущий размер меньше `minimumCapacity`
`void setLength (int newLength)`	Устанавливает длину строкового буфера в `newLength`, обрезая его или заполняя `null`-символами (`'\u0000'`)
`char charAt (int index)`	Возвращает символ с адресом `index`
`void getChars (int srcBegin, int srcEnd, char dst[], int dstBegin)`	Копирует символы из строкового буфера в `dst`: `srcBegin` — адрес начала области копирования; `srcEnd` — адрес конца области копирования; `dstBegin` — адрес начала области копирования в `dst`
`void setCharAt (int index, char ch)`	Изменяет символ по адресу `index` на `ch`
`StringBuffer append (char str[], int offset, int len)`	Добавляет в конец строкового буфера фрагмент массива символов `str[]`, начиная с `offset` длиной `len`
`StringBuffer append (<type> typevar)`	Добавляет в конец строкового буфера символьное представление переменной `typevar`. `<type>` может принимать значения `boolean`, `char`, `double`, `float`, `int`, `long`, `Object`, `string`. Если `<type> = char`, переменная может быть массивом `typevar[]`
`StringBuffer insert (int stroffset, char str[], int offset, int len)`	Вставляет в строковый буфер по адресу `stroffset` фрагмент массива символов `str[]`, начиная с `offset` длиной `len`
`StringBuffer insert (int stroffset, <type> typevar)`	Добавляет в строковый буфер по адресу `stroffset` символьное представление переменной `typevar`. `<type>` может принимать значения `boolean`, `char`, `double`, `float`, `int`, `long`, `Object`, `string`. Если `<type> = char`, переменная может быть массивом `typevar[]`

2.3. Функциональное программирование

Функциональное программирование — способ составления программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части — введение имени для функции и задание для этого имени выражения, вычисляющего значения функции, а единственным правилом композиции — оператор суперпозиции функции [18].

Функциональное программирование, как и другие модели неимперативного (непроцедурного) программирования, обычно применяется для решения задач, которые трудно сформулировать в терминах последовательных операций. Практически все задачи, связанные с искусственным интеллектом, попадают в эту категорию. Среди них следует отметить задачи распознавания образов, общение с пользователем на естественном языке, реализацию экспертных систем, автоматизированное доказательство теорем, символьные вычисления.

Функциональное программирование является одним из старейших. По происхождению оно тесно связано с лямбда-исчислением, изобретенным еще в начале 30-х годов XX в. логиком Алонзо Черчем совместно со Стивеном Клини. Для многих функциональная методология стала ассоциироваться с языком Lisp, созданным Джоном Маккарти в конце 50-х годов XX в.

2.3.1. Вычислительная модель

Функциональное программирование представляет собой одну из альтернатив императивному подходу. В императивном программировании алгоритмы — это описания последовательно исполняемых операций с использованием понятий «текущего шага исполнения» и «текущего состояния», которое меняется с течением времени. В функциональном программировании понятие времени отсутствует. Программы представляют собой выражения, а исполнение программ заключается в вычислении этих выражений.

В основе функционального программирования лежат структурирующие формы, помогающие рационально строить программу, реализующую некоторую функцию (т. е. выражать ее смысл простой и понятной комбинацией других функций) [18].

1. Композиция («*»). Применить результат композиции двух функций f и g — значит применить функцию f к результату применения функции g:

$$(f*g)(e) \rightarrow f(g(e)).$$

2. Общая аппликация («A») — применение указанной функции ко всем составляющим обрабатываемого предложения:

$$(Af)(e\ t) \rightarrow f(e)\ f(t).$$

3. Конструкция («,»). Применить результат конструкции двух функций f и g к аргументу e — значит, получить конкатенацию выражений $f(e)$ и $g(e)$:

$$(f,g)(e) \rightarrow f(e)\ g(e).$$

4. Редукция («/»). Идея редукции в том, что бинарная операция (двуместная функция) последовательно применяется, начиная с конца выражения вида $(x\ y\ t)$, т. е. выражения, в котором не меньше двух составляющих. Обрабатываемое выражение сворачивается (редуцируется) за счет последовательного «съедания» пар компонентов выражения, начиная с конца:

$$(/f)(x\ y\ t) \rightarrow (f)(x\ (/f)(y\ t)).$$

Следует отметить такие свойства функционального программирования, как *аппликативность* и *настраиваемость*.

1. *Аппликативность*: программа есть выражение, составленное из применения функций к аргументам. Программа состоит из совокупности определений функций, представляющих собой вызовы других функций, и предложений, управляющих последовательностью вызовов.

2. *Настраиваемость*: так как не только программа, но и любой программный объект (в идеале) является выражением, можно легко порождать новые программные объекты по образцу, как значения соответствующих выражений (применение порождающей функции к параметрам образца).

Настраиваемость активно используется в таком направлении программирования, как generic programming. Основная задача, решаемая в рамках этого направления — создание максимально универсальных библиотек, ориентированных на решение часто встречающихся подзадач (обработка агрегатных данных; потоко-

вый ввод-вывод; взаимодействие между программами, написанными на разных языках и различающимися в деталях семантики; универсальные библиотеки окон). Эти направления наиболее ярко представлены в STL — стандартной библиотеке шаблонов (контейнеров) языка Си++, а также — в реализации платформы .NET фирмы MicroSoft.

Нормальные алгоритмы Маркова

Базой для одной из моделей функционального программирования служат нормальные алгоритмы Маркова. Алгоритмическая схема Маркова (1954 г.) преобразует слова, заданные в некотором алфавите, на основе непосредственного доступа к различным частям слова. В этой схеме нет понятия ленты и передвижения по ней автомата (как в машине Тьюринга, описанной в п. 2.1). Алгоритм Маркова можно записать в следующем виде:

$$W_A^1 \left\{ \begin{matrix} \to \\ \mapsto \end{matrix} \right\} W_B^1$$

$$W_A^2 \left\{ \begin{matrix} \to \\ \mapsto \end{matrix} \right\} W_B^2$$

.....

$$W_A^n \left\{ \begin{matrix} \to \\ \mapsto \end{matrix} \right\} W_B^n$$

Здесь W_A^i — слово во входном алфавите;

W_B^i — слово в выходном алфавите;

$\left\{ \begin{matrix} \to \\ \mapsto \end{matrix} \right\}$ означает, что между словами может стоять одна из стрелок.

Таким образом, алгоритм Маркова (или *нормальный* алгоритм) представляет собой упорядоченный набор *формул подстановки* — пар слов, соединенных между собой стрелками двух видов:

$\to$ — замена левой части формулы на правую;

$\mapsto$ — замена левой части формулы на правую и останов (принудительное завершение работы алгоритма).

Выполнение нормального алгоритма для заданного слова происходит следующим образом.

Осуществляется поиск первой по порядку применимой формулы подстановки, т. е. формулы, левая часть которой содержится во входном слове.

Если такая формула найдена, то происходит замена во входном слове левой части формулы на правую и возврат в начало алгоритма. В том случае, когда во входном слове имеется несколько вхождений левой части, всегда заменяется первое (самое левое) вхождение.

Если очередная формула неприменима, то алгоритм переходит к проверке следующей формулы. Процесс выполнения нормального алгоритма завершается в одном из двух случаев:

- ни одна из формул более не применима к обрабатываемому слову;
- применилась формула, в которой левую и правую часть разделяет знак $\mapsto$.

В каждом из этих случаев считается, что нормальный алгоритм *применим* к данному входному слову. Если же в процессе выполнения *бесконечное число раз применяются незавершающие формулы, алгоритм считается неприменимым* к данному слову.

Рассмотрим несколько примеров нормальных алгоритмов.

По определению, для пустого слова нет специального обозначения. Кроме того, считается, что вхождения пустого слова имеются и справа, и слева от каждой буквы в преобразуемом слове. Таким образом, алгоритм

$\rightarrow a$

заменяющий пустое слово на букву a, будет бесконечно приписывать слева к входному слову букву a, следовательно, неприменим ни к одному слову.

Алгоритм

$\mapsto a$

напротив, применим к любому слову: припишет слева к входному слову букву a и остановится.

Алгоритм

$a \rightarrow 101$

$b \rightarrow 1001$

$c \rightarrow 10001$

осуществляет перекодировку слов из алфавита {*a*, *b*, *c*} в алфавит {0, 1} и применим к любому слову. Например, для входного слова *baabac* он будет выполняться в такой последовательности:

Преобразованное слово:	*Примененная формула подстановки:*
*b*101*abac*	1
*b*101101*bac*	1
*b*101101*b*101*c*	1
1001101101*b*101*c*	2
100110110011001101*c*	2
10011011011001101100001	3

Продемонстрируем возможности применения нормальных алгоритмов для арифметических вычислений. Для простоты будем изображать числа последовательностью единиц, например: число 4 — 1111; число 3 — 111.

Алгоритм получения частного и остатка от деления некоторого числа на 3 может быть записан следующим образом:

#111 → 1#
#↦#
→ #

Сначала будет применима только третья формула, которая припишет слева к заданному числу знак #. Далее, с помощью последовательного применения формулы 1, сформируется частное от деления. Вторая формула введена перед третьей для того, чтобы принудительно завершить вычисления. Например, для входного слова 1111111:

Преобразованное слово:	*Примененная формула подстановки:*
#1111111	3
1#1111	1
11#1	1
11#1	2 — останов

Языки функционального программирования

Язык функционального программирования (функциональный язык) — язык программирования, позволяющий задавать программу в виде совокупности определений функций.

В функциональных языках:

- функции обмениваются между собой данными без использования промежуточных переменных и присваиваний;
- переменные, однажды получив значение, никогда его не изменяют;
- циклы заменяются аппаратом рекурсивных функций.

Хронология появления основных языков функционального программирования:

- Lisp (1958 г.);
- РЕФАЛ(1968 г.);
- Scheme (1975 г.);
- FP(1977 г.);
- Miranda (1985 г.);
- ML(1985 г.);
- CaML (1985 г.);
- Haskell (1990 г.);
- Ocaml (Objective CaML — объектно-ориентированное расширение языка CaML, 1996 г.);
- Haskell 98 (объектно-ориентированное расширение языка Haskell, 1998 г.).

По этой хронологии заметен пик интереса к функциональным языкам (конец 70-х — начало 80-х годов XX в.). В настоящее время функциональное программирование перестало быть исключительно исследовательским. Современные языки функционального программирования (ML, Haskell) — строго типизированные, компилирующиеся — предназначены как для индивидуальной работы над программами, так и для коллективной разработки.

2.3.2. Язык функционального программирования РЕФАЛ

РЕФАЛ (**РЕ**курсивных **Ф**ункций **АЛ**горитмический язык) — один из старейших членов семейства языков функционального программирования. Впервые язык был реализован в 1968 г. в СССР и до сих пор используется. Создатель языка, Валентин Федорович Турчин, заложил в него концепции, намного опере-

дившие тогдашнее время. Как язык программирования, РЕФАЛ соединяет в себе математическую простоту с практической ориентацией на написание больших и сложных программ.

Лексика языка

При записи программы на языке РЕФАЛ используются прописные и строчные буквы латинского алфавита, цифры, знаки «-» («дефис»), «_» («подчеркивание») и специальные символы. Значение специальных символов приведено в табл. 2.7.

Таблица 2.7. **Специальные символы языка РЕФАЛ**

Символ	Значение
', "	Ограничители символов и символьных строк
(,)	Структурные скобки
<, >	Функциональные, или вычислительные, скобки
s, t, e	Индикаторы типа применения переменной (только строчные буквы)
=	«Заменить на»
;	«Иначе»
:	«Является» (знак сопоставления)
,	«Где»
.	«Далее следует индекс»
{	«Начало»
}	«Конец»
*	Строковые комментарии. Строка, которая начинается со звездочки, воспринимается как комментарий
/* */	Комментарии-вставки. Подстрока, ограниченная комбинациями, является комментарием

Используются также специальные системные ключевые слова: `$ENTRY` и `$EXTERNAL` (последнее можно использовать также в форме `$EXTERN` или `$EXTRN`). Системные ключевые слова записываются прописными буквами.

Идентификатор в языке РЕФАЛ — последовательность алфавитно-цифровых знаков, начинающихся с *прописной* буквы. Дли-

на идентификатора не должна превышать 15 знаков. В идентификаторах допустимы знаки дефис («-») и подчеркивание («_»), причем они являются эквивалентными. Строчные и прописные буквы внутри идентификаторов не различаются.

Данные и типы данных

Понятие переменной как контейнера, хранящего некоторое значение от одной операции присваивания до другой, в языке отсутствует. Данные представляют собой константы и результаты применения функций.

Данные могут быть следующих типов:

- символы и символьные строки;
- макроцифры;
- действительные числа.

Символы и символьные строки. Данные этого типа представляют собой последовательности символов, заключенные в кавычки. Могут использоваться как одинарные («'») так и двойные («"») кавычки, но открывающая и закрывающая кавычки должны быть одинаковы. Например, `'A'` есть символ, `'A+B'` есть символьная строка (последовательность трех символов).

Для представления в символьных строках самих кавычек необходимо их продублировать (если они находятся внутри строки, ограничиваемой того же вида кавычками). Так `'*,'''` является строкой из трех символов: звездочки, запятой и одинарной кавычки. Это может быть также записано с использованием двойных кавычек: `"*, '"`.

Макроцифры. Макроцифрами (в реализации РЕФАЛ-5) являются положительные целые числа в диапазоне от 0 до $2^{32} - 1$. Числа, превосходящие $2^{32} - 1$, могут быть составлены из макроцифр с использованим основания 2^{32} так же, как десятичные целые числа составляются из десятичных цифр. Для представления отрицательных целых чисел перед цифрами числа помещается знак «минус».

Например:

- 556 является символом-макроцифрой с численным значением 556;
- последовательность '-'25 воспринимается арифметическими функциями как число –25 (при записи программ знаки должны заключаться в кавычки);

- 1234 567 89 является последовательностью трех макроцифр, которая будет пониматься как $1234 \cdot 2^{64} + 567 \cdot 2^{32} + 89$.

Действительные числа. Действительные числа, не снабженные знаком, должны начинаться и заканчиваться цифрой и включать десятичную точку, либо букву E, либо и то и другое. Точный синтаксис действительных чисел может быть задан следующей последовательностью формул Бэкуса:

```
<Действительное число>::=<Действительное число без знака>|
                         <Знак числа><Действительное число без знака>|
<Действительное число без знака>::=
      <Последовательность цифр>.<Последовательность цифр>|
<Последовательность цифр>.<Последовательность цифр> E<Знак числа>
  <Последовательность цифр>|
  <Последовательность цифр> E <Знак числа><Последовательность цифр>
<Последовательность цифр>::=<Десятичная цифра>|
          <Последовательность цифр><Десятичная цифра>
<Знак числа> ::= +|–
```

Например:

```
215.73
-18E+15
0.003E-7
```

Переменные и выражения

Свободная переменная (или *переменная*) представляет собой индикатор типа применения, за которым следует точка, а за ней, в свою очередь, следует индекс:

```
<Переменная> ::= <Индикатор типа применения>.<Индекс>|
                 <Индикатор типа применения><Односимвольный индекс>
```

Индикаторами типа применения являются (обязательно строчные):

- s — переменная, представляющая один символ, макроцифру, идентификатор или действительное число;
- t – переменная, представляющая терм;
- e — переменная, представляющая выражение.

Индекс может быть идентификатором или макроцифрой. В случае односимвольного идентификатора или числа, представленного единственной цифрой, точка может быть опущена. Если

точка не опущена, идентификатор может начинаться со строчной буквы. Например:

- каждое из представлений `eX`, `e.X` и `e.x` означает один и тот же РЕФАЛ-объект;
- `t15` и `eSub` не являются правильными переменными.

Синтаксис РЕФАЛ-выражений следующий:

```
<Выражение> ::= <Пустое выражение>|<Терм> <Выражение>
<Терм> ::= <Символ>|<Символьная строка>|<Макроцифра>|
          <Действительное число>|<Переменная>|
          (<Выражение>)|<<Имя функции> <Выражение>>
<Имя функции> ::= <Идентификатор>
<Пустое выражение> ::=
```

Выражения бывают следующих типов:

- *объектное выражение* — последовательность конечного числа термов, где ни один терм не является свободной переменной. Объектное выражение не содержит вычислительных скобок. Структурные скобки следует объединять в пары согласно общепринятым правилам. Количество термов в выражении может быть нулевым, т. е. пустое объектное выражение (буквально ничего не содержащее) является допустимым;
- *выражение-образец* (или просто *образец*) представляет собой выражение, которое не включает никаких вычислительных скобок. Выражения-образцы отличаются от объектных выражений тем, что они могут включать свободные переменные;
- *общее выражение* — выражение, которое может содержать и свободные переменные, и скобки — как структурные, так и вычислительные.

Образец можно рассматривать как множество объектных выражений, которые могут быть получены в результате придания допустимых значений всем переменным. Так, образец `A e.1` представляет множество всех объектных выражений, начинающихся с символа `A`.

Всем вхождениям одной и той же переменной следует придавать одно и то же значение: образец `s.1 e.2 s.1` рассматривается как множество всех объектных выражений, которые начинаются и заканчиваются одним и тем же символом и содержат, по крайней мере, два символа.

Приведем некоторые примеры выражений и последовательностей, не являющихся выражениями:

A (A'+'B)'*'(C'-'D) (()'100'100() (()))	Объектные выражения
A (B)((C)) End	Не являются выражениями
'+' s.Free-var e.1 (e.Data) s1	Выражения-образцы

Операция сопоставления с образцом

Операция сопоставления с образцом обозначается как

```
E : P
```

где E — объектное выражение (*аргумент* операции);

P — образец;

«:» (двоеточие) — знак сопоставления.

Если существует подстановка S для переменных в P такая, что применение S к P приводит к E, то говорят, что сопоставление является *успешным*. Переменные в P принимают значения, предписываемые подстановкой S. В противном случае сопоставление считается *неуспешным*.

При выполнении распознавания аргумент E обозревается слева направо и выбирается первое успешное сопоставление E с P. В том случае, если имеется несколько способов присваивания значений свободным переменным в образце, при которых сопоставление может быть успешно, выбирается такой способ, при котором самая левая e-переменная принимает наиболее короткое значение.

Примеры некоторых сопоставлений приведены в табл. 2.8.

Таблица 2.8. **Примеры сопоставлений**

Сопоставление	Результат
`A B C : A e.1`	Успешно, если `e.1` принимает значение `B C`
`('ABC') '++' : s.X e.1`	Не может быть успешным, поскольку левая скобка не символ и не может быть сопоставлена с `s.X`
`('ABC')'++': (t.X) e.Out`	Успешно, когда `t.X` присваивается `'ABC'`, а `e.Out` присваивается `'++'`

Окончание табл. 2.8

Сопоставление	Результат
`'++' : s.1 e.2 s.1`	Успешно, когда `s.1` принимает значение `'+'`, а `e.2` — пустое выражения
`'+' : s.1 e.2 s.1`	Неуспешно
`AB'+'C'+'DEF : e.1'+'e.2`	Имеется более чем один способ придания значений для сопоставления. Согласно приведенному выше определению, символ `'+'` в образце будет идентифицироваться с первым символом `'+'` в аргументе (сопоставление слева-направо). Таким образом, `e.1` примет значение `AB`, а `e.2` — станет цепочкой `C'+'DEF`
`AB'-'(C'+'DEF): e.1'+'e.2`	Неуспешно, так как единственный символ `'+'` в аргументе находится внутри скобок. При сопоставлении его с `'+'` в образце необходимо было бы присвоить `e.1` и `e.2` такие значения, которые несбалансированы по скобкам, а это невозможно

Определение функции

Алгоритм с точки зрения языка РЕФАЛ является набором функций. Чтобы определить функцию, необходимо рассмотреть различные варианты ее аргумента и записать соответствующие предложения. Синтаксис определения функции следующий:

```
<Определение функции> ::= <Имя функции> {<Блок>}|
                          $ENTRY <Имя функции> {<Блок>}
<Имя функции> ::= <Идентификатор>
<Блок> ::= <Предложение>|<Предложение>;|<Предложение>; <Блок>
<Предложение> ::= <Левая часть>, <Список условий> = <Правая часть>|
                  <Левая часть> <Список условий>, <Блок окончания>
<Левая часть> ::= <Выражение-образец>
<Список условий> ::= <Условие>|<Условие>, <Список условий>
<Условие> ::= <Пустое условие>| <Аргумент>: <Выражение-образец>
<Аргумент> ::= <Общее выражение >
<Правая часть> ::= <Общее выражение>
<Блок окончания> ::= <Аргумент> : {<Блок>}
```

Таким образом, вычислительная часть функции представляет собой *блок* — перечень предложений, разделяемых точкой с запятой. Последнее предложение также может заканчиваться точкой с запятой. Порядок предложений в функциональном определении является существенным.

В соответствии с синтаксисом вызова функции вычислительные скобки могут быть вложенными.

Существуют ограничения на правую часть предложения:

- правая часть может включать только такие свободные переменные, которые входят также и в левую часть;
- индексы переменных должны быть уникальными, т. е. можно использовать `e.X` и `s.1` любое число раз, но если уже используется `e.X`, то запрещено использовать `s.X` или `t.X` в этом же предложении.

С формальной точки зрения все РЕФАЛ-функции являются функциями одного аргумента. Однако очень часто этот единственный аргумент состоит из частей, которые по существу являются подаргументами (но обычно называются просто *аргументами*). Когда определяется функция, на самом деле представляющая собой функцию нескольких аргументов, эти аргументы объединяются, чтобы образовать единственный формальный аргумент.

Для обозначения *вызова функции* используются угловые (вычислительные) скобки (<, >). Для выделения аргументов используются круглые скобки. Способ объединения задается *форматом* функций и может быть произвольным, например:

```
<F (e1)e2>
<F e1(e2)>
<F (e1)(e2)>
<F (e1)(e2)e3>
<F (e1)e2(e3)>
```

В формате для именования параметров можно использовать идентификаторы, например:

```
<F Graph (e.Gr) Start (e.St) Weights (eW)>
```

В РЕФАЛе присутствуют всего три синтаксические категории: символ, терм, выражение. Вследствие этого язык становится очень простым для программирования и формального анализа. Тем не менее язык позволяет имитировать в рамках своего простого синтаксиса многое из того, что достигается введением типов данных. Предположим, например, что нужно оперировать с рациональными числами, которые представлены тремя символами: знак, числитель, знаменатель. Для такой структуры данных можно ввести имя `Rat` и использовать ее в следующей форме:

```
(Rat s.Sign s.Numerator s.Denominator)
```

В этом случае нет необходимости связывать специальный тип с такой структурой данных, а произвольное рациональное число X записывается:

```
(Rat eX)
```

Произвольное отрицательное рациональное число, в свою очередь, может быть записано как:

```
(Rat '-'eX)
```

Для каждого типа данных легко можно определить функции доступа, которые предусматривают выделение структурных элементов, например:

```
Numerator {(Rat sS sN sD) = sN}
```

Однако очень часто использование функций доступа не нужно: прямое использование образцов делает определение и более кратким, и более ясным. Например, умножение рациональных чисел можно определить следующим образом:

```
MulRat {
  (Rat s.S1 s.N1 s.D1)(Rat s.S2 s.N2 s.D2) =
  (Rat <Mul-signs s.S1 s.S2> <* s.N1 s.N2> <* s.D1 s.D2>);
  }
```

где функция `Mul-signs` должна быть определена как функция вычисления знака результата. (Здесь оставлен открытым вопрос о том, как представить рациональное число 0) .

Встроенные функции

РЕФАЛ-5 содержит библиотеку стандартных встроенных функций. Она включает, например:

- арифметические функции;
- функции обработки символов и символьных строк;
- функции ввода-вывода.

Если имя, используемое для встроенной функции, присвоено функции, определяемой посредством РЕФАЛ, встроенная функция становится недоступной.

Арифметические функции. Основным форматом бинарной арифметической операции является следующий:

```
<ar-function (e.N1) e.N2>
```

Круглые скобки могут опускаться. Если аргументы — действительные числа, то проблемы не возникает, так как всякое такое число представлено непрерывной последовательностью знаков. Когда же первый аргумент представляет собой целое число, по умолчанию от него будет браться одна макроцифра, возможно, с предшествующим знаком, в то время как остальная часть поступит во второй аргумент.

Если оба аргумента арифметической функции являются целыми числами, результат представляет собой также целое число; в противном случае он является действительным числом. Набор арифметических функций представлен в табл. 2.9.

Таблица 2.9. **Встроенные функции языка РЕФАЛ**

Функция	Описание
Арифметические функции	
`<Add (e.N1) e.N2>` `<+ (e.N1) e.N2>`	Возвращает сумму операндов
`<Sub (e.N1) e.N2>` `<- (e.N1) e.N2>`	Вычитает `e.N2` из `e.N1` и возвращает разность
`<Mul (e.N1) e.N2>` `<* (e.N1) e.N2>`	Возвращает произведение операндов
`<Div (e.N1) e.N2>` `</ (e.N1) e.N2>`	Если по крайней мере один из аргументов является действительным, функция возвращает действительное частное. Если оба являются целыми, `Div` возвращает целое частное от деления `e.N1` на `e.N2`; остаток игнорируется. Возникает ошибка, если значением `e.N2` является 0
`<Divmod (e.N1) e.N2>`	Рассчитана на целые аргументы и возвращает `(e.Quotient) e.Remainder` `e.Quotient` — результат целочисленного деления, `e:Remainder` — остаток от деления. Остатку присваивается знак `e.N1` . Возникает ошибка, если значением `e.N2` является 0
`<Mod (e.N1) e.N2>`	Рассчитана на целые аргументы и возвращает остаток от деления `e.N1` на `e.N2` . Возникает ошибка, если значением `e.N2` является 0
`<Compare (e.N1) e.N2>`	Сравнивает два числа и возвращает '-', когда `e.N1` меньше, чем `e.N2`; '+', когда больше ; '0', когда числа равны
`<Trunc e.N>`	Возвращает усеченное целое число. `e.N` является целым числом

Продолжение табл. 2.9

Функция	Описание
`<Real e.N>`	Возвращает равное `e.N` действительное число. `e.N` является целым числом
`<Realfun (e.Function) s.N>` `<Realfun (e.Function) s.N1 s.N2>`	Возвращает значение функции `e.Function` одного или двух аргументов. `e.Function` должно быть строкой символов, которая является названием функции, имеющейся в языке C. Например, `<Realfun ('log') s.N>` возвращает логарифм `s.N`
Функции обработки символов и символьных строк	
`<Type e.Expr>`	Возвращает `s.Type e.Expr`, где `e.Expr` является неизменным, а `s.Type` зависит от типа первого элемента выражения `e.Expr`: `s.Type` `e.Expr` начинается с: `'L'` буквы; `'D'` цифры; `'F'` идентификатора или имени функции; `'N'` макроцифры; `'R'` действительного числа; `'O'` любого другого символа; `'B'` левой скобки; `'*'` `e.Expr` является пустым
`<Numb e.Digit-string>`	Возвращает макроцифру, представленную строкой `e.Digit-string`
`<Symb s.Macrodigit>`	Является обратной к функции `Numb`. Она возвращает строку десятичных цифр, представляющую `s.Macrodigit`
`<Implode e.Expr>`	Берет начальные алфавитно-цифровые символы выражения `e.Expr` и создает из них идентификатор (символическое имя). Начальная строка в `e.Expr` должна начинаться с буквы и заканчиваться неалфавитным символом, скобкой или признаком конца выражения. Строка не должна превышать 15 знаков. Подчеркивание и тире также допустимы. `Implode` возвращает идентификатор, за которым следует необработанная функцией часть выражения `e.Expr`. Если первый символ не является буквой, `Implode` возвращает макроцифру `0`, за которой следует аргумент
`<Explode s.Identifier>`	Возвращает строку символов, которая составляла `s.Idenitifier`
`<Chr e.Expr>`	Замещает всякую макроцифру в `e.Expr` знаком клавиатуры с таким же ASCII кодом по модулю 256

Продолжение табл. 2.9

Функция	Описание
`<Ord e.Expr>`	Является обратной к `Char`. Она возвращает выражение, в котором все символы замещены на макроцифры, совпадающие с символьными ASCII кодами
`<First s.N e.Expr>`	Разбивает `e.Expr` на две части — `e.1` и `e.2` — и возвращает `(e.1)e.2`. `s.N` является макроцифрой. Если исходное выражение `e.Expr` имеет по крайней мере `s.N` термов (на верхнем уровне структуры), то первые `s.N` термов поступают в `e.1`, а остальные — в `e.2`. В противном случае `e.1` совпадает с `e.Expr`, а `e.2` является пустым
`<Last s.N e.Expr>`	Подобна `First`, но `s.N` термов поступают в `e.2`
`<Lenw e.Expr>`	Возвращает длину выражения `e.Expr` в термах
`<Lower e.Expr>`	Возвращает исходное выражение `e.Expr`, в котором все прописные буквы замещены строчными буквами
`<Upper e.Expr>`	Подобна функции `Lower`. Все строчные буквы замещаются прописными
Функции ввода-вывода	
`<Card>`	Возвращает (замещается на) следующую строку из входного файла. Обычно она поступает с терминала, но ввод может быть переназначен средствами операционной системы. Возвращаемое выражение представляет собой последовательность набираемых символов (быть может, пустую). Байт признака конца файла в нее не включается. Если ввод производится из файла, при достижении конца файла возвращается макроцифра 0 (строк больше нет). Это используется в программах как индикатор конца ввода
`<Print e.Expr>`	Распечатывает выражение `e.Expr` на текущее выводное устройство и возвращает (заменяется на) `e.Expr`
`<Prout e.Expr>`	Распечатывает выражение `e.Expr` на текущее выводное устройство и возвращает пустое выражение
`<Open s.Mode s.D e.File-name>`	Открывает файл `e.File-name` и связывает его с файловым дескриптором `s.D`. `s.Mode` является одним из символов: `'w'`, `'W'` (открыть для записи) либо `'r'`, `'R'` (открыть для чтения). `e.File-name` может быть пустым: в этом случае будет попытка открыть файл `REFAL`*`descr`*`.DAT`, где *`descr`* является десятичным представлением дескриптора `s.D`. Если файла `e.File-name` не существует, то: — файл будет создан (в случае режима записи); — выдается ошибка (в случае режима чтение)

Окончание табл. 2.9

Функция	Описание
`<Get s.D>`	`s.D` является файловым дескриптором либо нулем. Действует подобно `<Card>`, за исключением того случая, когда получает входные данные из файла, указанного в `s.D`. Если ни один из файлов с таким файловым дескриптором еще не открыт, будет попытка открыть файл `REFAL`*`descr`*`.DAT`, где *`descr`* является десятичным представлением `s.D`. В случае неудачи возникает ошибочная ситуация, и выполнение программы завершается. Если значением `s.D` является 0, чтение будет проводиться с терминала
`<Put s.D e.Expr>`	`s.D` является файловым дескриптором либо нулем. Записывает `e.Expr` в файл с дескриптором `s.D` и возвращает `Expr` (подобно операции `Print`). Если ни один из файлов с таким файловым дескриптором еще не открыт для записи, откроется файл `REFAL`*`descr`*`.DAT`, где *`descr`* является десятичным представлением `s.D`. Если значением `s.D` является 0, осуществится вывод на терминал
`<Putout s.D e.Expr>`	Возвращает пустое значение (подобно `Prout`). Во всем остальном идентична функции `Put`
Функции работы со стеком	
`<Br e.Name'=' e.Expr>`	Помещает в стек выражение `e.Expr` под именем `e.Name`. Имя не должно содержать знак `'='`
`<Dg e.Name>`	Возвращает последнее выражение, сохраненное в стеке под именем `e.Name` и удаляет его из стека. Если не существует выражения, сохраненного под именем `e.Name`, `Dg` возвращает пустое выражение
`<Cp e.Name>`	Работает как `Dg`, но не удаляет выражение из стека
`<Rp e.Name'=' e.Expr>`	Замещает выражение, сохраненное в стеке под именем `e.Name`, на `e.Expr`
`<Dgall>`	Возвращает стек полностью

Функции обработка символов и строк. В табл. 2.9 приведен перечень функций, обеспечивающих обработку символов и символьных строк.

Ввод-вывод. Стандартный ввод и вывод, выполняемый посредством `Card` (клавиатура) и `Print` (печать), может быть переориентирован на другие файлы с помощью средств, обеспечи-

ваемых операционной системой. Для использования одного или более файлов ввода-вывода в дополнение к операциям с клавиатурой и экраном предназначены встроенные функции `Get` и `Put` в комбинации с функцией `Open`.

Файл может использоваться как в режиме чтения, так и в режиме записи. Один и тот же файл может быть использован поочередно в различных режимах. Функциям, которые работают с файлами, требуется в качестве аргумента файловый дескриптор. Дескриптор файла является макроцифрой в диапазоне 1—19. В некоторых операциях допустим дескриптор 0, который является ссылкой на терминал.

Чтобы использовать файл, его следует вначале *открыть* посредством функции `Open`.

После того как в файл произведена запись, он должен быть *закрыт*. Специальной РЕФАЛ-функции для закрытия файлов не существует, поэтому закрытие файлов происходит автоматически в двух случаях:

1) при завершении выполнения РЕФАЛ-программы;

2) когда канал, открытый для записи, переоткрывается (либо с этим же файлом, либо с другим).

Чтобы читать из файла, который открыт для чтения, активизируется:

```
<Get s.Channel>
```

Машина считывает одну строку символов из файла, связанного с `s.Channel`. Если она дошла до символа «Конец Файла», результатом будет число 0.

Чтобы записать строку в файл, открытый для записи, активизируется:

```
<Put s.Channel e.Expression>
```

Содержимое `e.Expression` будет возвращено в качестве значения и добавлено как строка к текущему содержимому файла, связанного с `s.Channel` .

Функция:

```
<Putout s.Channel e.Expression>
```

является аналогичной `Prout` (возвращает *empty*). Когда `Open` применяется для того, чтобы связать канал с файлом,

s.Channel не должна быть равной 0. При s.Channel, равной 0, функции Get, Put и Putout используют в качестве файла терминал (т. е. становятся эквивалентны Card, Print и Prout).

Для ввода РЕФАЛ-выражения используется функция Input. Синтаксис аргумента-выражения в этом случае в основном совпадает с общим синтаксисом выражения. Единственным отличием является менее строгое использование кавычек, потому что всегда читаются *объектные выражения*, а не РЕФАЛ-программы. Единственными нецифровыми набираемыми символами, которые не могут представлять сами себя, являются круглые скобки, пробелы и одинарные кавычки. Следует также разделять при наборе использование символа '-' как дефиса в идентификаторе и как знака действительного числа. Ослаблено также требование к первой букве идентификатора — она может быть как прописной, так и строчной.

Выражения, находящиеся для считывания в одном файле, разделяются пустыми строками. Пустая строка (дополнительный возврат каретки) будет также завершать ввод выражения с терминала.

Когда активизируется функция

```
<Input s.Channel>
```

РЕФАЛ-машина будет читать файл, связанный с s.Channel, до тех пор, пока не встретит пустую строку либо конец файла. Когда этот вызов активизируется снова, считывается следующее выражение, и т. д.

Функция Input может также быть использована просто вместе с именем файла, который нужно читать:

```
<Input e.File-name>
```

Автоматически будет найден свободный канал, ему будет присвоено имя файла e.File-name и последует вызов Open. Для чтения с терминала следует использовать <Input 0> или <Input>.

В табл. 2.9 приведены основные функции ввода-вывода.

Общий синтаксис РЕФАЛ-программы. РЕФАЛ-машина

РЕФАЛ-программа представляет собой перечень *функциональных определений* и *внешних объявлений функций*.

Общий синтаксис РЕФАЛ-программы следующий:

```
<РЕФАЛ-программа> ::= <Определение функции>|
   <Определение функции> <РЕФАЛ-программа>|
   <Определение функции>; <РЕФАЛ-программа>|
   <Внешнее объявление>; <РЕФАЛ-программа>|
   <РЕФАЛ-программа> <Внешнее объявление>;
<Внешнее объявление> ::= $EXTERNAL <Список имен функций>|
            $EXTERN <Список имен функций>|
            $EXTRN <Список имен функций>
<Список имен функций> ::= <Имя функции>| <Имя функции>,
                          <Список имен функций>
```

Для отделения одного внешнего объявления от другого должны использоваться точки с запятыми. Функциональные определения могут разделяться точкой с запятой.

Программы выполняет на языке РЕФАЛ абстрактное устройство, называемое РЕФАЛ-машиной.

РЕФАЛ-машина имеет два потенциально бесконечных хранилища информации — *поле программы* и *поле зрения*. Поле программы содержит РЕФАЛ-программу, которая загружается в машину перед запуском и не изменяется в ходе выполнения. Поле зрения (аналог пространства данных) хранит выражение без свободных переменных; такие выражения называются *определенными*. Поле зрения (т. е. выражение в этом поле) изменяется в ходе работы машины.

РЕФАЛ-машина работает в пошаговом режиме. Каждый шаг выполняется следующим образом. Если выражение в поле зрения не включает вычислительных скобок (т. е. является *пассивным*), РЕФАЛ-машина приходит к нормальному останову. В противном случае она выбирает в поле зрения одно из подвыражений вида `<F E>`, где `F` является именем функции, а `E` — выражением. Это подвыражение называется *первичным активным подвыражением*. Оно определяется как первое (слева) подвыражение `<F E>`, такое, что `E` пассивно, т. е. не содержит вычислительных скобок.

Первичное подвыражение `<F E>` трансформируется следующим образом. РЕФАЛ-машина сравнивает последовательно `E` с предложениями из определения `F`, начиная с первого, осуществляя поиск первого *применимого* предложения. Предложение применимо, если `E` может быть распознано как его левая часть `L`, т. е. сопоставление `E:L` является успешным.

После нахождения первого применимого предложения РЕФАЛ-машина копирует его правую часть R и применяет к ней подстановку, полученную при сопоставлении E:L. Таким образом, свободные переменные в R замещаются значениями, которые они должны принять для успешного сопоставления. Затем выражение, сформированное при этом, замещает первичное активное подвыражение <F E> в поле зрения. Этим завершается текущий шаг, и машина переходит к выполнению следующего шага. Если же применимых предложений в определении F не имеется, РЕФАЛ-машина приходит к аварийному останову: *'Отождествление невозможно'*.

Чтобы вычислить значение некоторой функции F с аргументом E (который должен быть объектным выражением), <F E> помещается в поле зрения РЕФАЛ-машины и машина запускается. Если после конечного числа шагов РЕФАЛ-машина приходит к нормальному останову, то содержимое поля зрения (также объектное выражение) является значением функции. Если же машина зацикливается или приходит к аварийному останову, значение функции считается неопределенным.

Порядок вычисления вложенных вызовов функций, используемый в РЕФАЛ-машине, называется *аппликативным* или порядком типа *изнутри-наружу*. Перед запуском вычисления любого вызова функции должны быть завершены все вычисления вызовов функций в аргументе.

Кроме функций, определяемых в РЕФАЛе посредством предложений, существует множество функций, которые не должны определяться в РЕФАЛе, но все-таки могут применяться, поскольку они *встроены* в систему. К этой категории, в числе прочих, относятся функции ввода-вывода, функции выполнения арифметических операций и функции работы с символьными строками (эти функции представлены выше).

Реализация РЕФАЛ-машины на компьютере отличается от абстрактной РЕФАЛ-машины тем, что перед просмотром определения функции в поле программы система проверяет, входит ли имя функции в список имеющихся встроенных функций. Если оно обнаружено в списке, активизируется соответствующая подпрограмма, которая выполняет требуемые операции и замещает вызов функции в поле зрения на его вычисленное значение.

Примеры функций

Рассмотрим некоторые примеры функций.

Функция удаления знаков «минус». Пусть необходимо иметь функцию, которая обрабатывает символьные строки и удаляет каждый знак '-'. Запись такого алгоритма как системы правил на естественном языке может быть следующей:

1. Взять первый символ аргумента. Если это `'-'`, удалить его. Далее применить трансформацию к оставшейся части аргумента, чтобы получить результат.

2. Если первый символ не `'-'`, оставить его без изменения и применить трансформацию к оставшейся части так же, как и в пункте 1.

3. Если строка пустая, результирующая строка также пуста. Конец работы.

Теперь выразим это на языке РЕФАЛ:

```
  Delm { /* Delm – функция удаления из строки знаков '-' */
/* Выделение первого знака и применение функции */
/* к оставшейся части выражения: */
       '-' e.1 = <Delm e.1>;
/* Пропуск первого знака, если это не '-', */
/* применение функции к оставшейся части выражения: */
       s.2 e.1 = s.2 <Delm e.1>;
/* Если строка пустая: */
       =  ;
  }
```

Приведем последовательность вычислений для случая, когда аргументом `Delm` является `'a-b+c-+d'` :

```
<Delm 'a-b+c-+d'>
'a'<Delm '-b+c-+d'>
'a'<Delm 'b+c-+d'>
'ab'<Delm '+c-+d'>
'ab+'< Delm 'c-+d'>
'ab+c'< Delm '-+d'>
'ab+c'< Delm '+d'>
'ab+c+'< Delm 'd'>
'ab+c+d'< Delm >
'ab+c+d';
```

Такой алгоритм требует последовательного анализа каждого отдельного символа строки. Лучшее решение — за один шаг РЕФАЛ-машины найти первый символ `'-'` и удалить его. Если

в строке нет ни одного символа '-', она трансформируется в самое себя:

```
Delm {
        e.1 '-' e.2 = e.1 <Delm e.2>;
        e.1 = e.1; }
```

При том же аргументе, что и выше, вычисление проводится всего за три шага:

```
<Delm 'a-b+c-+d'>
'a'<Delm 'b+c-+d'>
'ab+c'<Delm '+d'>
'ab+c+d'
```

Функция вычисления факториала. Определим функцию ! (факториал) для целых неотрицательных чисел (с использованием арифметических функций).

Рекурсивный алгоритм вычисления факториала реализуется следующей последовательностью предложений:

1. Задание значения 0! = 1.

2. Вычисление факториала на основе последовательного формирования выражения $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$.

В языке РЕФАЛ не существует прерываемых или отложенных вызовов функций, так как работа РЕФАЛ-машины состоит просто из последовательности замещений. Но функциональный вызов, как единое целое, может ожидать своей очереди в поле зрения.

Рекурсивное определение факториала в терминах РЕФАЛ следующее:

```
Fact {
  0 = 1;
  sN = <* sN <Fact <- sN 1>>>; }
```

Приведем вычисление <Fact 3> :

```
<Fact 3>
<* 3 <Fact <- 3 1>>>
<* 3 <Fact 2>>
<* 3 <* 2 <Fact <- 2 1>>>>
<* 3 <* 2 <Fact 1>>>
<* 3 <* 2 <* 1 <Fact <- 1 1>>>>>
<* 3 <* 2 <* 1 <Fact 0>>>>
<* 3 <* 2 <* 1 1>>>
<* 3 <* 2 1>>
<* 3 2>
6
```

Когда функция вычисляется, первые $n \cdot 2$ шагов порождают структуру:

```
<* n· <* n-1 <* n-2 ...*<Fact 0>> ... >>
```

затем работает первое предложение из определения `Fact`, и последующие шаги умножения приводят к результату. Здесь отсроченные в результате рекурсии вызовы обозначены символом «*», а не `Fact`.

Функция перевода слов. Иногда для решения задачи вводятся функции, которые не имеют аргументов, а возвращают некоторые множества значений. Пусть необходимо переводить русские слова на английский язык. Для этого введем функцию `Dictionary`, которая в качестве значения будет возвращать пары — слово и его перевод.

Такая функция-таблица, например, для нескольких русских слов может иметь вид:

```
Dictionary { =  (('комар') 'gnat')
                (('муравей') 'ant')
                (('муха') 'fly')
                (('оса') 'wasp')
                (('паук') 'spider')
                (('пчела') 'bee') }
```

Далее введем функцию `TransWord` от двух аргументов, которую будем вызывать в формате:

```
<TransWord (e.Word) e.Table>
```

Скобки отделяют первый аргумент (слово, которое нужно перевести) от второго аргумента — таблицы, которая при этом используется.

При словаре-таблице заданного формата определение функции `TransWord` имеет вид:

```
/* Поиск и перевод слов из словаря */
TransWord {
  (e.Word) e.1 ((e.Word) e.Trans) e.2 = e.Trans;
  (e.Word) e.1  =  '***';
  }
```

Второе предложение говорит, что если слово не найдено в таблице, то в перевод подставляется три звездочки.

Теперь можно определить функцию перевода Trans через TransWord:

```
/* Перевод слов с русского языка на английский */
Trans {
  e.W = < TransWord (e.W) <Dictionary>>; }
```

Основные приемы программирования

Встраивание алгоритмов в функции. Если определены функции F1, F2, F3, то их последовательное применение к объекту выражается следующим образом:

```
F { eX = <F3 <F2 <F1 eX>>> }
```

То есть результат одной функции становится аргументом для другой.

Когда различные части объекта должны подвергнуться обработке по отдельности, определяются функции, которые применяются к соответствующим частям, например:

```
F { (eX)(eY)eZ = <F1 eX> <F2 eY> <F3 eZ> }
```

Взаимодействие между функциями в РЕФАЛ-системе может производиться либо через поле зрения, либо посредством прямых вызовов. В примере последовательной обработки, приведенном выше:

```
F {eX = <F3 <F2 <F1 eX>>> }
```

используется взаимодействие через поле зрения. Каждая из трех функций F1, F2 и F3 ничего не знает о других. Она оставляет свое значение в поле зрения, а следующая функция принимает его. Альтернативой этому является определение F через систему прямых вызовов (F сама просто вызывает F1):

```
F { eX = <F1 eX> }
```

Разбиение выражения на части. В результате выполнения некоторой последовательности шагов можно получить структурированный объект, различные части которого подразумевают различные применения. Например, функция деления нацело </s.N1 s.N2> продуцирует комбинацию (s.Quotient)

s.Remainder. Если необходим только остаток от деления s.N1 на s.N2, можно использовать вызов:

```
<Rem </ s.N1 s.N2>>
```

где Rem определяется предложением:

```
Rem { (sQ) eR = eR }
```

Ветвление по значению функции. В том случае, когда возникает потребность вычислить некоторую функцию F, а затем в зависимости от результата продолжать вычисления тем или иным путем, вводится вспомогательная функция F1, которая анализирует результат F и проводит требуемый выбор:

```
<F1 <F eX>>
```

Скобки как указатели. Предположим, необходимо написать программу, которая удаляет все лишние пробелы в данной строке символов, т. е. замещает каждую группу подряд стоящих пробелов единственным пробелом. Назовем функцию, выполняющую эту работу, Blanks. Очевидно следующее решение: как только обнаруживается пара соседних пробелов, следует удалить один из них и повторять это до тех пор, пока имеются такие пары. В результате получается предложение (знак '_' обозначает пробел):

```
e1'__'e2 =  <Blanks e1'_'e2>
```

Если это предложение оказывается неприменимым, желаемый результат достигнут, и работа завершается с помощью предложения:

```
e1 = e1
```

Таким образом, определение функции имеет вид:

```
Blanks {
     e1'__'e2 = <Blanks e1'_'e2>;
     e1 = e1; }
```

Проанализируем алгоритмическую эффективность такой программы. Переменная e1 в первом предложении является *открытой*. Первоначально она принимает пустое значение, а затем удлиняется до тех пор, пока не будет найдена первая комбина-

ция '__' (если таковая имеется). Переменная e2 является *закрытой*, поэтому оставшаяся часть аргумента не просматривается.

На следующем шаге работы РЕФАЛ-машины весь аргумент Blanks, включая проекцию e1, будет просмотрен снова при поиске пары смежных пробелов. Это, очевидно, является потерей времени, так как проекция e1 не может содержать такую пару. Программу можно скорректировать, вынося e1 за пределы вычислительных скобок в правой части. Определение приобретает вид:

```
Blanks {
     e1'__'e2 = e1 <Blanks '_'e2>;
     e1 = e1; }
```

Когда вычисляется функция Blanks, левая скобка вызова функции используется как указатель, который отделяет обработанную часть строки от части, еще не подвергавшейся обработке.

Рассмотрим следующий пример, где скобки вызова функции не могут использоваться подобным образом. Допустим, желательно определить корректирующую функцию Correct, уничтожающую в заданной строке символов всякий символ, за которым следует знак уничтожения '#'. Если в строке имеется несколько знаков уничтожения, функция ликвидирует соответствующее число предшествующих им символов. Так, если при наборе строки замечена ошибка, можно продолжить:

```
Кантр####онтраст цветов создавал особый ка#олорит
```

и если Correct применяется к этой строке, результатом будет:

```
Контраст цветов создавал особый колорит
```

Прямое определение такой функции имеет вид:

```
Correct {
     e1 sA'#' e2  =  <Correct e1 e2>;
     e1  =  e1; }
```

Согласно этому алгоритму аргумент будет просматриваться с начала столько раз, сколько в нем встречается символов уничтожения. Но в этом случае, чтобы сделать алгоритм эффективнее, нельзя просто вынести строку e1 за пределы скобок вызова функции, так как ее правый конец может еще нуждаться в коррекции.

В такой ситуации указателями могут служить структурные скобки. В самом деле, с каждой круглой скобкой должен быть связан адрес сопряженной ей скобки, так что возможен мгновенный переход от одной скобки к другой. По отношению к функции Correct это означает, что следует заключать в скобки начало строки, которое уже просмотрено. Первоначальным аргументом должно быть:

```
()Кантр####онтраст цветов создавал особый ка#олорит
```

После первого использования знака уничтожения аргумент приобретет вид:

```
(Кант)###онтраст цветов создавал особый ка#олорит
```

и т. д. Правая скобка здесь по существу является указателем.

Если желательно сохранить формат функции Correct таким, каким он был определен ранее, т. е. просто <Correct e.String>, то следует ввести вспомогательную функцию, скажем, Cor, в формате:

```
<Cor (e.Scanned) e.NotScanned>
```

Эффективно будет работать следующее определение:

```
Correct {
   e1 = <Cor () e1>; }

  Cor {
/* Если признак удаления находится сразу за скобкой – */
/* удалить последний символ внутри скобок и знак удаления; */
      (e1 sX) '#'e2 = <Cor (e1) e2>;
/* Если признак удаления находится не сразу за скобкой – */
/* удалить его вместе с предыдущим символом и */
/* заключить левую часть выражения (до удаленного символа) */
/* в скобки */
   (e1) e2 sX '#'e3 = <Cor (e1 e2) e3>;
   /* Если нет признаков удаления */
   (e1) e2 = e2; }
```

Изменение формата. Для того чтобы определить рекурсивный процесс, часто требуются дополнительные подаргументы функции. В этом случае определяется вспомогательная функция с требуемым форматом. В приведенном выше примере было видно, что функция Cor определена для того, чтобы определить функцию Correct.

Расширенный РЕФАЛ

Рассмотрим некоторое расширение базового РЕФАЛа, которое делает программирование более легким, а программы — более наглядными. Средствами расширения являются:

- where-конструкции, или условия;
- with-конструкции, или блоки;
- функции работы со стеком.

Условия. *Where-конструкция* (или *условие*) накладывает дополнительное условие на применимость предложения. Она следует за левой частью предложения, отделяясь от него специальным *where-with знаком* РЕФАЛа. В языке РЕФАЛ-5 этот знак может быть представлен либо амперсантом «&», либо запятой «,» (используется также и для выделения with-конструкций).

Условие является сопоставляемой парой, где образцом (правым операндом) может служить любое выражение-образец, а аргументом (левым операндом) может являться любое РЕФАЛ-выражение. Аргумент при этом должен включать только те переменные, которые имеют определенные значения на момент проверки условия (*связанные* переменные). Для условия, которое непосредственно следует за левой частью предложения, это требование означает, что в его аргументе могут использоваться только те переменные, которые появляются в левой части предложения. Выражение-образец в правой части условия может включать как связанные переменные, так и *свободные* переменные, которые еще не были определены и получают значения в процессе сопоставления.

В предложении могут встретиться несколько последовательных where-конструкций. Условия, введенные таким образом, будут вычисляться (проверяться) в заданной последовательности.

Пусть задано условие `E : P`. Оно вычисляется следующим образом. Во-первых, РЕФАЛ-машина порождает новое поле зрения, помещает `E` в это поле и заменяет переменные из `E` их значениями. Затем машина работает, как обычно, над этим полем зрения до тех пор, пока его содержимое не станет пассивным. Когда процесс завершается, результат сопоставляется с образцом `P`. При этом сопоставлении те переменные в `P`, которые являются уже ранее связанными, заменяются на свои значения. Свободные переменные принимают значения в процессе сопоставления.

После того как проверка условия завершается успехом либо неудачей, временное поле зрения, созданное для `E`, уничтожает-

ся, и РЕФАЛ-машина возвращается к полю зрения, из которого вызывалось E. Если сопоставление является успешным, вычисляется следующее условие. Если условий больше нет — левая часть предложения заменяется его правой частью. Если сопоставление терпит неудачу — предложение неприменимо.

Каждый образец в последовательности условий может присваивать значения новым переменным, которые становятся связанными в последующих условиях. В конце концов, все эти переменные получат значения и смогут быть использованы в правой части предложения.

Рассмотрим функцию PreAlph, которая устанавливает отношение предшествования между символами в смысле обычного алфавитного порядка:

```
PreAlph {
  /* Если буквы совпадают, */
  /* отношение предшествования принимает значение "истина": */
  s.1 s.1 = T;
  /* Если буквы различны — */
  /* результат вычисляется с помощью функции Before: */
  s.1 s.2 = <Before s.1 s.2 In <Alphabet>>; }

Before {
/* Если буквы расположены в порядке, заданном
/* функцией Alphabet - */
/* значение "истина" */
       s.1 s.2 In e.A s.1 e.B s.2 e.C = T;
/* иначе — значение "ложь" */
       e.Z = F; }

Alphabet { = 'abcdefghijklmnopqrstuvwxyz'; }
```

Это определение записано средствами базового РЕФАЛа. Переопределим функцию PreAlph, используя Where-конструкцию расширенного РЕФАЛа (в этом случае можно не вводить вспомогательную функцию Before):

```
PreAlph {
   s.1 s.1 = T;
   /* Введено условие <Alphabet>: e.A s.1 e.B s.2 e.C  */
   s.1 s.2, <Alphabet>: e.A s.1 e.B s.2 e.C = T;
   e.1 = F;  }
```

При проверке условия вычисляется значение функции <Alphabet>. В последующем процессе сопоставления s.1 и s.2 заменяются их значениями, т. е. символами, подлежащими срав-

нению. Сопоставление завершится успешно тогда и только тогда, когда `s.1` предшествует `s.2` в алфавите.

Блоки. With-конструкции позволяют использовать *блоки* (напомним, что блоком является список предложений в фигурных скобках) прямо в рамках функционального определения без введения для этой цели вспомогательной функции.

В качестве примера with-конструкции, рассмотрим функцию упорядочения пар:

```
Order {
       (e.1)e.2 = <Order1 <Pre (e.1)(e.2)> (e.1)e.2>;}

Order1 {
         T (e.1)e.2 = e.1 e.2;
         F (e.1)e.2 = e.2 e.1;
         };
```

Предикат `<Pre (e1)(e2)>` выражает некоторое отношение предшествования (порядка), заданное на строках символов и принимает значение `T`, если строка `e1` предшествует строке `e2`, и `F` — в противном случае.

Вспомогательная функция становится ненужной, если используется полный РЕФАЛ:

```
Order {
  (e.1)e.2, <Pre (e.1)(e.2)>:
            { T = (e.1)(e.2);
            F = (e.2)(e.1);
            };
  }
```

With-конструкция начинается таким же образом, как и where-конструкция — с where-with-знака. Затем следует общее РЕФАЛ-выражение, использующее в аргументе только связанные переменные, и двоеточие — знак операции сопоставления. Однако вместо образца в with-конструкции используется блок.

Фигурные скобки, объединяющие группу предложений в блок, имеют двоякое значение:

1) дают возможность сопоставлять аргумент с несколькими образцами: в приведенном примере `<Pre (e.1)(e.2)>` вычисляется однажды, после этого РЕФАЛ-машина пытается сопоставить его последовательно с левыми частями в блоке, а замещение выполняется как обычно;

2) выполняют функцию завершения процесса сопоставления в предшествующей левой части и в условиях: когда управление переходит к левой фигурной скобке, нет пути назад к удлинению e-переменных. Какие бы значения ни были присвоены переменным вне блока, они должны такими и остаться. В случае, когда ни одно из предложений неприменимо, последует аварийное завершение.

На самом деле блок является функциональным определением, но эта функция не имеет присвоенного имени и вызывается по *аргументу* (выражению, следующему за where-with-знаком). Можно видеть, что блок в определении функции `Order` в точности тот же, что и во вспомогательной функции `Order1`. With-конструкция позволяет определить безымянную функцию именно в том месте, где она нужна.

Рассмотрим следующее определение:

```
Fun {
        e.1'+'(e.2)e.3, <P e.2>: T = (e.2);
        e.1 = <Prout 'No such term'>;
};
```

Эта функция находит первый заключенный в скобки терм после '+' такой, что выражение внутри удовлетворяет предикату `P` . Если предикат определен как:

```
P  { s.1'*'s.2 = T;
  e.X = F;  }
```

то вычисление функции:

```
<Fun  A-B+(C/D)+(C*D)>
```

приводит к результату (`C*D`).

Если заключить условие и правую часть в фигурные скобки так, что они образуют блок:

```
Fun { e.1'+'(e.2)e.3>, <P e.2>: { T = (e.2) };
   e.1 = <Prout 'No such term'>;  }
```

то алгоритм следует читать следующим образом: найти первый заключенный в скобки терм, следующий за '+', а затем проверить, удовлетворяет ли заключенное в скобки выражение предикату `P`. Если теперь вычисляется тот же вызов, результатом будет аварийный останов «Распознавание невозможно», так как первым термом после '+' является (`C/D`), и `P` не удовлетворяется.

В случае такого определения функции не будет проведено попытки удлинить e.1: можно только *войти* в фигурные скобки, окружающие блок, но никогда нельзя выйти из них.

Работа со стеком. РЕФАЛ является строго функциональным языком. Но в качестве уступки более традиционным способам программирования, в нем еще остается ряд встроенных функций, которые позволяют пользователю употреблять присваивания.

Если реализация некоторого алгоритма требует поддерживать глобальные параметры или таблицы, не передавая их как часть аргумента от одного функционального вызова к другому, то можно сохранить такие параметры в стеке. Функция, сохраняющая значение параметра в стеке, вызывается следующим образом:

```
<Br Name '=' E >
```

где *Name* — имя, которое должно быть присвоено параметру; E — выражение.

Извлечь значение параметра из стека можно посредством вызова:

```
<Dg Name>
```

Функция, сохраняющая значение в стеке, при вычислении исчезает из поля зрения, но на ее месте остается выражение E в реальной связной структуре, представляющей поле зрения. Значение выражения связывается с именем *Name* и сохраняется как строка (*Name '=' Value*).

Функция <Dg *Name*> при вычислении замещается сохраненным выражением без просмотра последнего. Значение выражения при этом извлекается из стека.

Для работы со стеком определена также функция копирования Cp, которая имеет такой же формат и действие, как и Dg, но с ее помощью сохраненное выражение копируется, а не извлекается.

При многократном использовании Br и Dg порождают стек сохраненных значений для каждого имени. Функция Br помещает в стек каждое следующее значение для данного имени, не затрагивая предыдущих значений. Функция Dg извлекает последнее сохраненное значение. Таким образом, эти функции работают как стеки значений для каждого используемого имени. Однако в отличие от строгой семантики стеков, если стек для

данного имени пуст, результатом Dg является пустое значение, т. е. он не является неопределенным, как это должно было бы быть для обычного стека.

Предположим, создается программа, которая имеет дело с некоторыми объектами и время от времени порождает новые объекты того же вида. Им присваиваются последовательные номера, начиная с 1. Функция, которая порождает новые объекты, должна иметь доступ к первому свободному индексу. Он может быть сохранен в стеке под некоторым именем, например 'freeIndex'. Тогда при запуске программы выполняется:

```
<Br 'freeIndex=' 1>
```

Как только понадобится породить новый индекс, в качестве индекса используется функция:

```
<Next 'freeIndex'>
```

Функция Next получает число, сохраненное в стеке под ее аргументом-именем, и сохраняет в стеке значение, увеличенное на 1:

```
Next {
      e.Name, <Dg e.Name>: e.Value  = e.Value
           <Br e.Name'=' <+ e.Value 1>>; }
```

Рассмотрим программу, транслирующую строки итальянских слов в строки слов английских. Функция Dictionary была применена для хранения трансляционной таблицы (словаря):

```
Dictionary { =  (('комар') 'gnat')
                (('муравей') 'ant')
                (('муха') 'fly')
                (('оса') 'wasp')
                (('паук') 'spider')
                (('пчела') 'bee') }
```

Вместо хранения этой таблицы в качестве правой части предложения можно было бы хранить ее как выражение под одним и тем же именем, скажем, 'dict'.

Выражения, сохраняемые с помощью Br, хранятся в форме двусвязных списков, как и все выражения в поле зрения. Однако сохраненное выражение готово к применению, в то время как правая часть предложения, подобная той, которая используется для Dictionary, будет прочитана РЕФАЛ-машиной посимволь-

но в процессе создания соответствующего двусвязного списка в поле зрения. Поэтому, когда существенны пространственные ограничения, предпочтительнее сохранять таблицу в правой части предложения, но если более важно время выполнения, таблицу следовало бы сохранять в стеке:

```
<Br 'dict=' (('комар') 'gnat')
            (('муравей') 'ant')
            (('муха') 'fly')
            (('оса') 'wasp')
            (('паук') 'spider')
            (('пчела') 'bee')>
```

Одним из преимуществ сохранения таблиц является то, что они могут быть легко обновлены во время выполнения программы.

Таким образом, стек является строкой термов вида

```
(e.Name'=' e.Value)
```

При добавлении терма строка растет слева.

Функции работы со стеком представлены в табл. 2.9.

2.4. Логическое программирование

Логическое программирование появилось в конце 60-х годов XX в. Логика позволяет сделать заключение об истинности или ложности одних утверждений, исходя из знаний об истинности или ложности других. При создании языков программирования положения традиционной логики стали использовать сравнительно недавно. В классическом процедурном подходе к программированию центральным понятием, характеризующим класс задач, считается алгоритм их решения. Однако любой осмысленный класс задач характеризуется, прежде всего, определенными знаниями о фактах, понятиях, соотношениях и связях в предметной области, задающей этот класс. Суть логического подхода заключается в том, что машине в качестве программы предлагается не алгоритм, а формальное описание предметной области и решаемой проблемы (функции) в виде аксиоматической системы. Тогда поиск решения сводится к логическому выводу в этой системе аксиом посредством компьютера.

Логическое программирование — подход, согласно которому программа содержит описание проблемы в терминах утверждений (фактов и логических формул) об объектах и их взаимодействии в предметной области, а решение проблемы система выполняет с помощью встроенных механизмов логического вывода. При логическом программировании всегда можно получить результаты работы программы, точно не зная, как на самом деле система их нашла. Поэтому различают два уровня смысла логической программы — декларативный и процедурный. Декларативный смысл определяет, *что* должно быть результатом работы программы, а процедурный смысл — *как* этот результат был получен.

Способность системы логического программирования прорабатывать многие процедурные детали самостоятельно делает возможным рассматривать декларативный смысл программы относительно независимо от ее процедурного смысла. Задача программиста при этом заключается в грамотном представлении предметной области в виде системы логических формул и такого множества отношений на ней, которое наиболее полно описывает решаемую проблему.

Логическое программирование пережило пик популярности в середине 80-х годов XX в., когда оно было положено в основу проекта разработки программного и аппаратного обеспечения вычислительных систем пятого поколения.

2.4.1. Вычислительная модель

Логическое программирование основано на формальной логике. Программа задается как набор логических утверждений. Система находит решение задачи путем применения операций *унификации* (сопоставления) и *редукции* (преобразования, упрощения). Искомые величины задаются в виде переменных в логических отношениях и запросах.

Переменные в логическом программировании (так же, как и в функциональном) представляют собой лишь символическое обозначение некоторого объекта и значения их во времени не меняются (в отличие от переменных в понимании процедурного программирования). Значение переменной может быть либо найдено, либо нет.

Дизъюнкт Хорна

В логике теории задаются при помощи аксиом и правил вывода. В логическом программировании аксиомы принято называть фактами, а правила вывода ограничиваются по форме до так называемых «дизъюнктов Хорна».

Дизъюнкт Хорна — это дизъюнкт с не более чем одним неотрицательным членом, т. е. логическое предложение, имеющее следующий общий вид записи:

$$P_0 \vee \neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n, \quad n \geq 0,$$

где P_i — атомарная формула (предикат).

Эквивалентной формой представления дизъюнкта Хорна является запись вида:

$$P_1 \,\&\, P_2 \,\&\, \ldots \,\&\, P_n \rightarrow P_0.$$

Формула P_0 при этом называется *целевым дизъюнктом* или *целью*.

Для доказательства утверждений используются методы унификации и резолюций.

Метод унификации

Унификация — это сопоставление двух произвольных термов, содержащих переменные, с целью определения того, можно ли присвоить этим переменным такие значения, чтобы получились два одинаковых терма.

Например, унификация термов `f(X, 2)` и `f(1, Y)`, где `X`, `Y` — переменные, выдаст подстановку: `X = 1, Y = 2`. в этом случае говорят, что переменные *конкретизированы*.

Метод резолюции

Метод резолюции — метод доказательства теорем, использующий правило вывода, называемое *резолюцией*.

Резолюция заключается в следующем: если выводимы дизъюнкты

$$P \vee Q \quad \text{и} \quad \neg P \vee R,$$

где P — атомарная формула, а Q и R обозначают остальные части дизъюнктов (возможно пустые), то выводим и дизъюнкт

$$Q \vee R.$$

Выводимый дизъюнкт называется *резольвентой.*

Метод резолюций основан на последовательном доказательстве отдельных утверждений, входящих в посылку дизъюнкта Хорна, для доказательства его следствия. То есть, применение метода резолюций к правилу

```
a :- b, c.
```

приведет к последовательному доказательству утверждений `b` и `c`.

Факты (они же аксиомы) представляются как правила с пустой «посылкой».

Языки логического программирования

Языки логического программирования содержат конструкции, позволяющие выполнить описание проблемы в терминах фактов и логических формул, а собственно решение проблемы выполняет система с помощью механизмов логического вывода.

Родоначальником языков логического программирования является Prolog (1971 г.). В 80-х годах XX в. появилось целое семейство ему подобных: Prolog II (1980), IC-Prolog (1982), Mprolog, T-Prolog, Concurrent Prolog, PARLOG (1983), Goedel (1992), Mercury (1993).

Класс задач логического программирования практически совпадает с классом задач функционального программирования.

Области применения языков логического программирования:

- создание интерфейсов для общения с ЭВМ на естественном языке;
- перевод с одного языка на другой;
- разработка компиляторов;
- решение задач теории графов;
- создание экспертных систем;
- создание динамических реляционных баз данных;
- создание пакетов символьных вычислений.

2.4.2. Язык программирования Prolog

Prolog — язык логического программирования, основанный на логике дизъюнктов Хорна, был создан Аланом Колмероэ в 1971 г. Будучи декларативным языком программирования, Prolog

воспринимает в качестве программы некоторое описание задачи и сам производит поиск решения, пользуясь методом резолюций.

Основными свойствами языка являются:

- встроенный механизм сопоставления с образцом;
- механизм вывода с поиском и возвратом;
- иерархические структуры данных;
- естественная рекурсия.

В настоящее время (наряду с большим количеством «исследовательских» версий) существует несколько «промышленных» реализаций языка Prolog. «Промышленный» транслятор, как правило, порождает исполняемый код, сопоставимый по эффективности с кодом аналогичной программы на императивных языках.

Лексика языка

При записи программы на языке Prolog используются прописные и строчные буквы латинского алфавита, цифры, знак «_» («подчеркивание») и следующие специальные символы:

```
'  "  <  >  +  *  -  /  :  ;  .  ,  =  &  ~  !  @  #  $  (  )  ?
```

Для ввода в программу комментариев используются специальные скобки, если комментарий занимает несколько строк:

```
/* Скобки комментария,
 занимающего несколько строк */
```

и знак «%», если комментарий занимает одну строку:

```
%  Комментарий от знака до конца строки
```

Для обозначения переменных, символьных констант, объявлений типов и предикатов используются идентификаторы (имена).

Имя может начинаться с любой латинской буквы или символа подчеркивания («_»), затем следует любая комбинация букв, цифр и символа «_».

При образовании имен необходимо учитывать следующие правила:

- имена строковых констант должны начинаться со строчной буквы;
- имена переменных должны начинаться с прописной буквы или символа подчеркивания.

Типы данных

Типы данных в Прологе называются *доменами*. Домены подразделяются на системные и пользовательские:

<Домен> ::= <Системный домен> | <Пользовательский домен>

Системные типы данных. Перечень системных типов, определенных в языке, следующий:

<Системный домен> ::= `integer` | `string` | `symbol` | `char` | `real`

Из них к простым системным типам данных относятся:

- `char` — отдельный символ, заключенный в апострофы;
- `integer` — целое число (диапазон изменения значений зависит от реализации, например от –32 768 до 32 767);
- `real` — действительное число в обычной или экспоненциальной форме записи (диапазон также зависит от реализации).

К системным структурированным типам данных относятся символьные строки:

- `symbol` — символьная константа, имеет две формы записи:
  - последовательность букв, цифр и знаков подчеркивания, которая начинается со строчной буквы;
  - последовательность символов, заключенная в двойные кавычки;
- `string` — последовательность символов, заключенная в двойные кавычки (символьная строка).

Объявление новых (нестандартных) типов данных (доменов) возможно с использованием стандартных и ранее объявленных:

<Пользовательский домен> ::= <Список имен доменов> = <Домен>
<Список имен доменов> ::= <Имя домена> | <Имя домена>,
<Список имен доменов>

Примеры объявления типов данных:

```
year = integer
age  = integer
name = symbol
title = string
```

Структуры

Структурный объект (структура) состоит из нескольких элементов, каждый из которых в свою очередь может быть структу-

рой. При этом структура в программе ведет себя как единый объект. Элементы объединяются в структуру с помощью *функторов*. Синтаксис объявления функтора следующий:

<Функтор> ::= <Имя функтора> | <Имя функтора >(<Список доменов>)
<Список доменов> ::= <Домен> | <Домен>, <Список доменов>

Простая структура представляет собой функтор, а составная — перечень функторов — возможных альтернатив:

```
<Структура> ::= <Функтор> | <Функтор>; <Структура>
```

Например, объявление структур «Точка на плоскости», «Отрезок» и «Треугольник» может быть таким:

```
point_type = point(real, real)
line_type = line(point_type, point_type)
triangle_type = triangle(point_type,point_type,point_type)
```

Таким образом, структурные объекты можно изображать в виде деревьев. Корнем дерева служит функтор, а ветвями — компоненты. Если компонент является функтором, то дерево продолжает расти вниз. На рис. 2.4 представлено дерево функтора `line` с конкретными значениями аргументов.

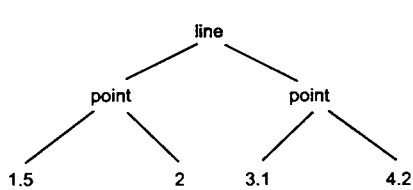


Рис. 2.4. Дерево функтора `line`

Ссылки на доменную структуру осуществляются по имени левой части, например, для составной структуры два следующих объявления эквивалентны:

```
figure_type = line_type; triangle_type
figure_type = line(point_type, point_type);
        triangle(point_type, point_type, point_type)
```

Такой метод структурирования данных позволяет рекурсивно описывать объекты сложных типов.

Переменные и константы

При программировании на языке Prolog переменные не нуждаются в объявлении.

Константы могут быть любого из стандартных типов языка. Значения констант должны быть записаны:

- со строчной буквы, исключая кириллицу (символьные константы):

```
fact1
summa
person
```

- в одинарных кавычках (отдельный символ) или двойных кавычках (строковая константа):

```
'c'
"summa="
"сумма"
```

- последовательностью цифр, символа e и знаков «+», «-», «.», если они являются целыми или вещественными числами:

```
25
0.5
3.2e-4
```

Переменная имеет имя, состоящее из букв, цифр и символа подчеркивания. Имя переменной всегда *начинается с прописной буквы* или *с символа подчеркивания*:

```
X_y
Summa
List_of_figures
_x2
```

Переменная может принадлежать одному из стандартных типов данных или пользовательскому типу, определенному в секции описания типов данных **domains**. Можно использовать так называемую *анонимную переменную* (т. е. переменную без имени), которая записывается в виде одного символа подчеркивания ('_').

Синтаксически все объекты данных языка Prolog представляют собой *термы*:

<Терм> ::= <Константа> | <Переменная> | <Структура>

Предложения языка

Помимо описания типов данных (которое может отсутствовать) программа на языке Prolog содержит *предложения*. Каждое предложение заканчивается точкой. Предложения бывают трех

типов: факты, правила и запросы (цели). Для записи предложений используются *предикаты*.

Предикаты. Предикаты служат для объявления как данных, так и правил их обработки. При объявлении предиката указываются типы данных аргументов. Аргументами предикатов являются термы:

<Предикат> ::= <Имя предиката> (<Список доменов>)

Например, могут быть описаны предикаты «муж» и «жена»:

```
husband(symbol)
vife(symbol)
```

Каждому из объявленных предикатов соответствуют предложения (дизъюнкты) языка.

Факты. Факты содержат утверждения, которые являются всегда верными (истинными)

<Факт> ::= <Имя предиката> (<Список термов>)
<Список термов> ::= <Терм> | <Терм>, <Список термов>

Таким образом, факт — это предикат с конкретизированными аргументами, например:

```
husband("Bob")
vife("Ann")
```

Правила. Правила в свою очередь содержат утверждения, истинность которых зависит от некоторых условий:

<Правило> ::= <Голова дизъюнкта> : - <Список условий>
<Голова дизъюнкта> ::= <Предикат>
<Список условий> ::= <Предикат> | not (<Предикат>) |
<Предикат> <Логическая операция> <Список условий>
<Логическая операция> ::= , | and | ; | or

Форма `not (<Предикат>)` означает отрицание предиката и допустима только в правой части правила.

Предикаты в списке условий могут связываться логическими операциями: конъюнкцией («`,`», «`and`») и дизъюнкцией («`;`», «`or`»). Общий результат правой части правила вычисляется в соответствии с заданной логической формулой. Переменные предикатов связаны квантором всеобщности.

Например, правило:

```
exist(X):-husband(X); vife(X); child(X).
```

читается следующим образом: для всех X, если X является мужем, *или* женой, *или* ребенком, то X — член семьи.

Правило «быть сестрой», например, может быть записано так:

```
sister(Sister, Brother) :-  female(Sister),
                            parents(Sister, Father, Mother),
                            parents(Brother, Father, Mother).
```

Это означает: для всех Sister и Brother, если они имеют общих родителей и Sister — женщина, то Sister является сестрой Brother.

Цели. С помощью целей формулируются вопросы к системе о том, какие из утверждений являются истинными. Цель представляет собой предикат с аргументами-переменными, например, предикат:

```
husband(X)
```

несет в себе вопрос: «Кто является мужем?».

Предикат цели может иметь форму `not (<Предикат>)`.

Структура программы

Программа на языке Prolog включает в себя нескольких программных секций, каждой из которых предшествует ключевое слово (табл. 2.10).

Таблица 2.10. **Заголовки программных секций в Prolog**

Наименование	Описание
Constants	Объявление и задание используемых в программе констант
Domains	Определение имен и структур (типов данных) используемых объектов
Database	Объявление предикатов базы данных
Predicates	Объявление предикатов (отношений между объектами)
Clauses	Определение фактов и правил
Goal	Объявление цели решения

В программе не обязательно наличие всех секций. Обычно представлены разделы Predicates и Clauses.

Если в программе описан раздел `Goal`, то цели, перечисленные в разделе, называются *внутренними*. Если цель задается при вызове программы, она является *внешней*, и раздел `Goal` при этом отсутствует. В программе может использоваться только одна внешняя цель. Раздел `Goal` обычно записывают в конце программы.

Ключевые слова названий разделов могут содержать как прописные, так и строчные буквы.

***Секция* `Domains`.** Секция используется для объявления имен и структур программных объектов. Объявления в секции `domains` бывают трех типов.

1. Объявление нестандартных типов данных через стандартные или объявленные ранее, например:

```
age, number = integer
```

Это предложение объявляет два домена целого типа, которые используются для объектов, синтаксически схожих, а семантически различных.

2. Объявление списка элементов:

```
mylist = element*,
```

где `mylist` — объявление списка элементов, `element` — элемент, ранее описанный пользователем в `Domains` либо один из стандартных типов Prolog, «*» обозначает список. Например,

```
nmblist = integer*
```

объявляет список целых чисел.

3. Объявление структуры, состоящей из нескольких простых или сложных объектов:

```
region = functor1(d1,d2,...); functor2(d3,d4,...);...
```

где `region` объявляет структуру; `functor1`, `functor2` — имена функторов (альтернатив) составной структуры; `d1`, `d2`, ..., `d3`, `d4` — типы данных, стандартные или определенные ранее в программе в секции `Domains`.

***Секция* `Predicates`.** В этой секции перечисляются все предикаты со своими областями определения аргументов.

Возможны предикаты с одинаковым именем. В этом случае они могут иметь различное число аргументов, и варианты для каждого предиката объявляются отдельно.

Секция **`Database`.** Объявление оперативной базы данных. Синтаксис объявления такой же, как и в секции `Predicates`. Объявленные здесь предикаты не должны объявляться в секции `Predicates`, но дизъюнкты этих предикатов могут присутствовать в секции `Clauses`.

Секция **`Clauses`.** Секция содержит описания предложений (дизъюнктов) для всех предикатов. В этой секции описываются факты и правила вывода.

Факты и правила, имеющие в качестве заголовка один и тот же предикат, должны следовать в программе непосредственно друг за другом. Последовательности фактов и правил с одинаковым заголовком представляют собой *процедуру*.

Секция **`Goal`.** Секция содержит *внутренний* запрос к программе. Для такого запроса осуществляется поиск только *первого* подходящего решения, причем вывод как результата, так и информации о неуспехе должен организовать программист. Целей может быть несколько. В этом случае они перечисляются через запятую, например:

```
Goal
     sister(X, "Миша"),
     write("Сестра Миши - "),
     write(X).
```

Стандартные предикаты

Для решения реальных задач в язык были введены элементы, лежащие за пределами чистой логики. Такие элементы реализованы как стандартные (встроенные) предикаты.

Большинство стандартных предикатов выполняет несколько функций в зависимости от состояния параметров, входящих в предикат. Известные к моменту обращения к предикату параметры называются входными, неизвестные — выходными. Работу предиката определяет соотношение входных и выходных параметров, которое называется поточным шаблоном.

В общем случае для любого предиката можно построить 2^n поточных шаблона (n — число параметров). Например, если предикат будет вызываться с двумя параметрами, можно сформировать четыре варианта поточного шаблона:

(i,i), (i,o), (o,i), (o,o),

где i — входной параметр; o — выходной параметр.

Однако не для каждого предиката все возможные варианты поточного шаблона имеют смысл.

В табл. 2.11 приводятся некоторые наиболее часто употребляемые стандартные предикаты, сгруппированные по их функциональному назначению.

Таблица 2.11. **Стандартные предикаты языка Prolog**

Предикат	Действие
Предикаты ввода-вывода	
`readln(StringVariable)` `(string) – (o)`	Считывает строку с текущего устройства ввода и связывает ее с заданной переменной `StringVariable`. Обычно чтение производится с клавиатуры. В качестве конца строки используется символ возврата каретки
`readint(IntgVariable)` `(integer) – (o)`	Читает целое число с текущего устройства ввода и связывает его с заданной переменной. Обычно текущим устройством ввода является клавиатура
`readreal(RealVariable)` `(real) – (o)`	Читает действительное число с текущего устройства чтения и связывает его с заданной переменной `RealVariable`. Обычно чтение производится с клавиатуры
`readchar(CharVariable)` `(char) – (o)`	Читает символ с текущего устройства ввода и связывает его с заданной переменной `CharVariable`. В отличие от `inkey` устанавливает режим ожидания ввода
`inkey(CharVariable)` `(Char) – (o)`	Читает символ со стандартного устройства ввода. В отличие от предиката `readchar` выполнение программы не прерывается. Поэтому `inkey` применяют главным образом для организации циклов ожидания
`Keypressed`	Выполняется успешно, если нажата некоторая клавиша. В отличие от предиката `inkey` с помощью `keypressed` можно установить, нажата ли клавиша, не читая при этом введенный с клавиатуры символ
`write(Variable|Constant)`	Выполняется запись заданных значений переменных и констант в заданное активное окно на текущем устройстве вывода
`nl`	Вызывает возврат каретки и перевод строки

Продолжение табл. 2.11

Предикат	Действие
Предикаты обработки строк	
Предикаты обработки строк используются для разделения строк либо на список отдельных символов, либо на список заданных групп символов. Лексема — это последовательность символов, имеющих смысл. Она определяется либо как имя в соответствии с синтаксисом Prolog, либо как строчное представление числа, при этом знак возвращается отдельно, либо как отдельный символ	
`frontchar(String,FrontChar, RestString)` `(string,char,string)-(i,o,o) (i,i,o) (i,o,i) (i,i,i) (o,i,i)`	Разделяет заданную строку `String` согласно поточному шаблону на две части: первый символ `FrontChar` и оставшаяся часть строки `RestString`
`fronttoken(String,Token, RestString)` `(string,string,string)-(i,o,o) (i,i,o) (i,o,i) (i,i,i) (o,i,i)`	Разделяет строку, заданную параметром `String`, на лексему `Token` и остаток `RestString` согласно поточному шаблону
`frontstr(Lenght,Inpstring, StartString,RestString)` `(integer,string,string,string)-(i,i,o,o)`	Разделяет строку `Inpstring` на две части. `StartString` будет иметь длину `Lenght` первых символов исходной строки, `RestString` представляет собой остаток строки `InpString`
`concat(String1,String2,String3)` `(string,string,string)-(i,i,o) (i,o,i) (o,i,i) (i,i,i)`	Слияние строк, согласно поточному шаблону, по формуле: `String3 = String1 + String2`
`str_len(String,Length)` `(string,integer)-(i,i) (i,o) (o,i)`	Определяет длину `Length` строки `String`
`isname(StringParam)` `(string)-(i)`	Завершается успешно, если `StringParam` есть имя, удовлетворяющее синтаксису языка
Предикаты преобразования типов	
`char_int(CharParam,IntgParam)` `(char,integer)-(i,o) (o,i) (i,i)`	Преобразует символ в код ASCII согласно поточному шаблону
`str_int(StringParam,IntgParam)` `(string,integer)-(i,o) (o,i) (i,i)`	Строка, представляющая целое десятичное число, преобразуется в это число
`str_char(StringParam,CharParam)` `(string,char)-(i,o) (o,i) (i,i)`	Один знак, заданный как строка, преобразуется в символ
`str_real(StringParam,RealParam)` `(string,real)-(i,o) (o,i) (i,i)`	Строка, представляющая действительное десятичное число, преобразуется в это число

Продолжение табл. 2.11

Предикат	Действие
Предикаты работы с базой данных	
`consult(FileName)` `(string) – (i)`	Добавляет текстовый файл с именем `FileName` к текущей базе данных. Этот файл содержит факты. Выполнение предиката не будет успешным, если в файле имеются синтаксические ошибки. Текстовый файл может быть, например, создан в результате выполнения предиката `save`
`save(DosFileName)` `(string) – (i)`	Записывает динамическую базу данных на диск в текстовый файл с именем `FileName`. Для хранения каждого факта используется отдельная строка. После этого файл можно снова загрузить в оперативную память, используя предикат `consult`
`asserta( Term )` `(DbaseDom) – (i)`	Заносит факт `Term` в базу данных перед другими фактами. Факт должен быть термом, принадлежащим области определения `DbaseDom`
`assertz( Term )` `(DbaseDom) – (i)`	Заносит факт `Term` в конец базы данных. Факт должен быть термом, принадлежащим области определения `DbaseDom`
`retract( Term )` `(DbaseDom) – (_)`	Удаляет первый факт из базы данных, который соответствует заданному факту `Term`
`retractall(Term)` `(InternalDatabaseDomain)– (_)`	Очищает базу данных
Управляющие предикаты	
`Exit`	Выполняет немедленный выход из программы
`Fail`	Вынуждает завершиться предикат ложно и, следовательно, возвратиться к предыдущей точке разветвления (`backtracking`)
`True`	Значение предиката всегда истинно
`!`	Отсечение (прекращение перебора между головой дизъюнкта и данным знаком)

Окончание табл. 2.11

Предикат	Действие
Прочие стандартные предикаты	
`random(RealVariable)` `(real)-(o)`	Равномерное псевдослучайное число в диапазоне от 0 до 1
`random(MaxValue,RandomInt)` `(integer,integer)-(i,o)`	Равномерное псевдослучайное целое число `RandomInt` в диапазоне от 0 до `MaxValue`
`findall(Variable,Atom,` `ListVariable)`	В списке `ListVariable` возвращаются все решения для переменной `Variable` в предикате `Atom`
`not(Atom)`	Выполняется успешно, если заданный предикат `Atom` представляет собой цель, которая не достигается
`free(Variable)`	Выполняется успешно, если `Variable` не является конкретизированной переменной
`bound(Variable)`	Выполняется успешно, если `Variable` является конкретизированной переменной

Арифметические и логические операции

Язык Prolog рассчитан главным образом на обработку символьной информации, при которой потребность в арифметических вычислениях обычно мала. Средствами для таких вычислений являются арифметические выражения. В арифметических выражениях могут участвовать операнды (числа и переменные), арифметические операции, скобки, стандартные функции. Набор арифметических и логических операций полностью соответствует наборам операций в табл. 1.1 и 1.3 гл. 1. Приоритет выполнения операций соответствует общему приоритету, определенному в п. 1.2 гл. 1. В состав алгебраических и тригонометрических функций входят: `sin(X)`, `cos(X)`, `tan(X)`, `arctan(X)`, `ln(X)` — натуральный логарифм, `log(X)` — десятичный логарифм, `abs(X)`, `exp(X)`, `sqrt(X)` и т. п.

Организация вычислительного процесса

Система Prolog обладает встроенным механизмом логического вывода, благодаря этому от пользователя требуется только описание своей задачи с помощью аппарата логики предикатов первого порядка, а поиск решения система берет на себя.

Основным механизмом поиска решения задачи при этом является *сопоставление*. Такой механизм позволяет решать широкий класс задач просто путем перебора всех возможных состояний.

Если получен запрос (т. е. цель, которую нужно удовлетворить), Prolog-система пытается определить его истинность двумя способами:

- во-первых, цель успешно удовлетворяется (т. е. считается истинной), если она сопоставляется с существующим фактом (так как факты всегда являются истинными);
- во-вторых, цель считается истинной, если она сопоставляется с головой некоторого правила, и условия, входящие в правую часть правила, могут быть выполнены успешно.

В случае успешного сопоставления выдается ответ **yes**, т. е. цель согласована.

Если попытка сопоставления в базе знаний завершается неудачей, но при этом остаются альтернативные (еще не применявшиеся) правила или факты, будет осуществляться возврат и попытка использовать эти альтернативы. Если все альтернативы закончились неудачно, то считается, что начальная цель неудовлетворительна. В этом случае выдается отрицательный ответ **(No solutions)**.

В качестве аргументов цель может содержать константы или переменные. Переменная представляет собой не конкретизированную величину, которую Prolog должен конкретизировать, т. е. выполнить ее подстановку константой. После того, как цель, содержащая переменные, будет согласована, выдаются значения, которыми будут заменены переменные, входящие в цель.

Так как в языке отсутствует оператор присваивания, а знаком равенства обозначается операция логического сравнения, запись вида `N = N + 1` представляет собой заведомо ложное утверждение: величина никогда не равна самой себе, увеличенной на единицу. В таком случае надо использовать другую переменную: `N1 = N + 1`.

Программа на языке Prolog не содержит глобальных переменных, т. е. одни и те же имена переменных можно употреблять во многих правилах данной программы, при этом между такими переменными будет отсутствовать какая-либо связь.

Таким образом, Prolog-система, отвечая на запросы, выполняет процедуру вычисления списка целей с учетом заданной программы. «Вычислить» цели означает попытаться достичь их,

а процедура «Вычислить» может быть определена следующими входом и выходом (рис. 2.5):

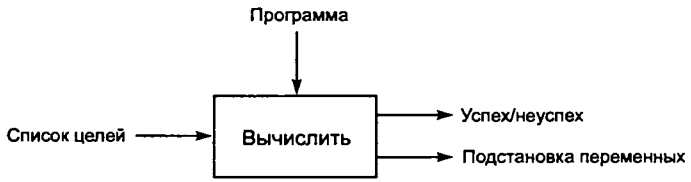


Рис. 2.5. Входы и выходы процедуры вычисления списка целей

Вход — программа и список целей.

Выход — признак успех/неуспех и подстановка переменных. Признак успех/неуспех принимает значение «да», если все цели достижимы, и «нет» — в противном случае. Подстановка переменных порождается только в случае успешного завершения.

Рассмотрим пример Prolog-программы:

```
domains
animal = string
predicates
big(animal)
small(animal)
brown(animal)
black(animal)
gray(animal)
dark(animal)
goal
dark(X), big(X)
clauses
/* факты */
big("медведь").      % Предложение 1
big("слон").         % Предложение 2
small("кот").        % Предложение 3
brown("медведь").    % Предложение 4
black("кот").        % Предложение 5
gray("слон").        % Предложение 6
/* правила */
dark(Y) :- black(Y). % Предложение 7: любой черный
                     % объект является темным
dark(Y) :- brown(Y). % Предложение 8: любой коричневый
                     % объект является темным
```

Процедура «Вычислить» для списка целей такой программы выполняет следующую последовательность шагов вычислений.

1. Исходный список целей: `dark(X), big(X).`

2. Просмотр программы сначала и поиск предложения, у которого голова сопоставима с первым целевым утверждением `dark(X)`. Найдена формула 7:

```
dark(Y) :- black(Y).
```

Замена первого целевого утверждения конкретизированным телом предложения 7 — порождение нового списка целевых утверждений: `black(X), big(X).`

3. Просмотр программы для нахождения предложения, сопоставимого с `black(X)`. Найдено предложение 5: `black("кот").` Это предложение является фактом, поэтому список целей после конкретизации сокращается до

```
big("кот")
```

4. Просмотр программы в поисках цели `big("кот")`. Ни одно предложение не найдено. Поэтому происходит возврат к шагу 3 и отмена конкретизации `X = "кот"`. Возвращаемся к списку целей: `black(X), big(X).`

Просмотр программы продолжается с предложения, следующего после 5. Ни одно предложение не найдено. Происходит возврат к шагу 2 и продолжение просмотра программы с предложения, следующего после 7. Найдено предложение 8:

```
dark(Y) :- brown(Y).
```

Первая цель конкретизируется и заменяется на `brown(X)`. Теперь список целей выглядит так:

```
brown(X), big(X).
```

5. Просмотр программы для обнаружения предложения, сопоставимого с `brown(X)`. Найдено предложение `brown("медведь")`. Список целей сокращается до

```
big("медведь").
```

6. Просмотр программы и обнаружение предложения `big("медведь")`. Это предложение является фактом, поэтому список целей становится пустым. Это указывает на успешное завершение, а соответствующая конкретизация переменных такова:

```
X = "медведь".
```

Рассмотрим еще одну задачу логического вывода:

Бутси — коричневая кошка.

Корни — черная кошка.

Мак — рыжая кошка.

Флэш, Ровер, Спот — собаки.

Ровер — рыжая.

Спот — белая.

Все животные, которыми владеют Том и Кейт, имеют родословные.

Том владеет всеми черными и коричневыми животными.

Кейт владеет всеми собаками небелого цвета, которые не являются собственностью Тома.

Алан владеет Мак, если Кейт не владеет Бутси и если Спот не имеет родословной.

Флэш — пятнистая собака.

Определить, какие животные не имеют хозяев.

Решение задачи может быть получено с помощью следующей Prolog-программы:

```
Predicates
        cat(symbol)
        dog(symbol)
        color(symbol, symbol)
        have(symbol, symbol)
        rod(symbol)
        animal(symbol)
clauses
        cat(butsi).    cat(korni).    cat(mac).
        dog(rover).    dog(fles).     dog(spot).
        color(butsi,brown).
        color(korni,black).
        color(mac,yellow).
        color(rover,yellow).
        color(spot,white).
        color(fles,black_and_white).
        have(tom,X):-color(X,black); color(X,brown).
        have(keit,X):-dog(X), not(color(X,white)),
         not(have(tom,X)).
        have(alan,mac):-not(have(keit,butsi)), not(rod(spot)).
        rod(X):-animal(X), have(tom,X).
        rod(X):-animal(X), have(keit,X).
        animal(X):-cat(X); dog(X).
goal animal(X),not(have(_,X)),write(X).
```

Ответ на вопрос программы изображен на рис. 2.6.

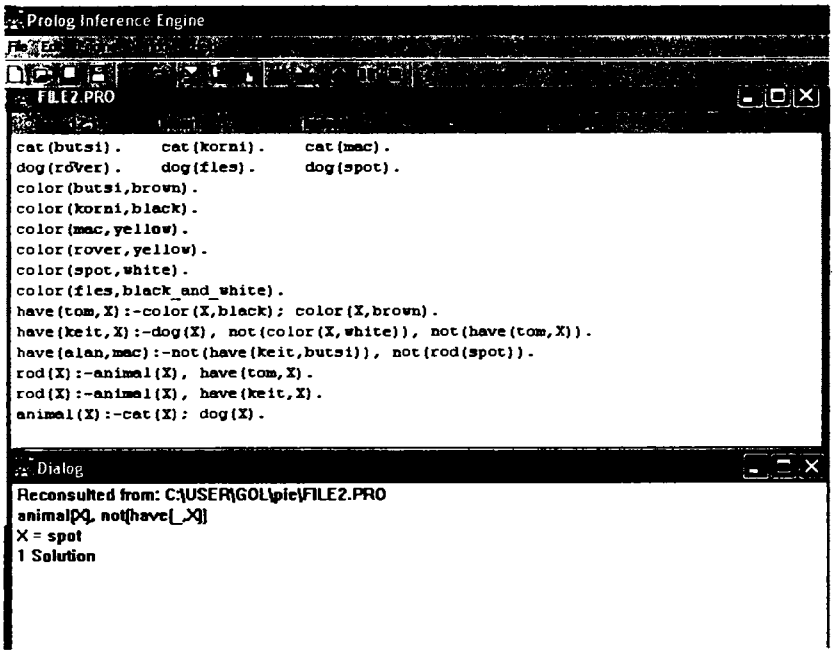


Рис. 2.6. Иллюстрация работы Prolog-системы

Рассмотрим пример использования структур при построении простой базы данных «Семья».

В базе данных о семьях каждая семья может описываться одним предложением. Будем считать, что каждая семья состоит из трех объектов: мужа, жены и детей. Поскольку количество детей в разных семьях может быть разным, то объект «дети» целесообразно представить в виде списка, состоящего из произвольного числа элементов. Каждого члена семьи в свою очередь можно представить структурой, состоящей из четырех компонентов: имя, фамилия, дата рождения и сведения о работе. Сведения о работе — это значение «не работает» или указание места работы и оклада (рис. 2.7).

Для описания такой базы данных введем следующие структуры, описывающие, соответственно, дату рождения (число, месяц, год) и сведения о работе:

```
Domains
data=dat(integer, symbol, integer)
work=worker(symbol, integer); notwork
```

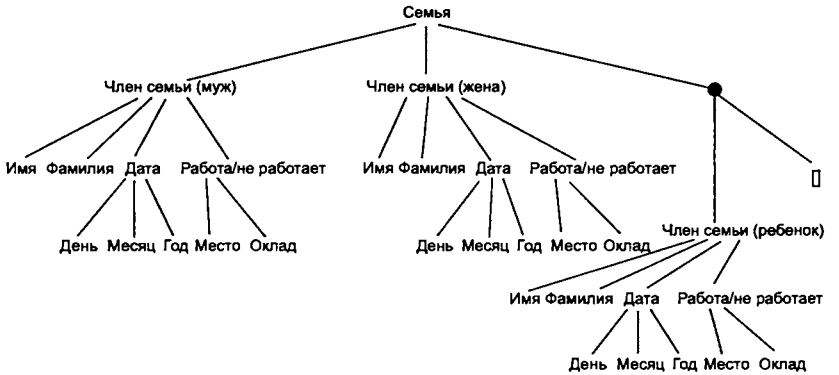


Рис. 2.7. Структурированная информация о семье

Тогда основная структура, соответствующая описанию каждого члена семьи, может выглядеть таким образом:

```
memfamily = mem(symbol, symbol, data, work)
```

Известную информацию о семьях (факты) можно занести в базу данных с помощью следующих предложений:

```
family(mem(jon,kelly,dat(17,may,1970),worker(ibm,15000)),
  mem(bony,kelly,dat(29,may,1971),notwork)),
  [mem(pat,kelly,dat(5,april,1993),notwork),
  mem(liz,kelly,dat(10,april,2000),notwork)]).
family(mem(bob,rob,dat(14,may,1968),worker(bbc,12000)),
  mem(sally,rob,dat(5,october,1970),worker(lotus,11000)),[ ]).
  % Нет детей
```

Теперь к базе данных можно формировать запросы. Так, всех членов семьи kelly можно задать с помощью терма:

```
goal family(mem(_,kelly,_,_),_,_).
```

Символы подчеркивания обозначают различные анонимные переменные, значения которых нас не интересуют.

Можно найти все семьи с двумя детьми с помощью выражения:

```
goal family(X,_, [_,_]).
```

Чтобы найти имена и фамилии всех женщин, имеющих трех детей, можно задать вопрос:

```
goal family(_, mem(Name,Fam,_,_),[_,_,_]).
```

Главным моментом в этих примерах является то, что указывать интересующие нас объекты можно не только по их содержимому, но и по их структуре, т. е. иметь результат в виде целой структуры или в виде отдельных элементов структур.

Можно также создать набор процедур, который делал бы взаимодействие с базой данных более удобным. Такие процедуры являлись бы частью пользовательского интерфейса. Вот некоторые из них:

```
husband(X):-family(X,_,_).                % X - муж
vife(X):-family(_,X,_).                   % X - жена
child(X):-family(_,_,Children), member(X,Children).% X - ребенок
exist(X):-husband(X); vife(X); child(X). % X - любой член семьи
dohod(mem(_,_,_,worker(_,D)),D).      % D - доход работающего
dohod(mem(_,_,_,notwork),0).          % 0 - доход неработающего
```

Этими процедурами можно воспользоваться, например, в следующих запросах к базе данных:

1. Найти имена и фамилии всех людей из базы данных:

```
Goal exist(mem(Nam,Fam,_,_)).
```

2. Найти всех работающих жен:

```
Goal vife(mem(Nam,Fam,_,worker(_,_))).
```

3. Найти фамилии людей, чей доход меньше 8000:

```
Goal exist(Man), dohod(Man,D), D<8000.
```

Управление процессом вычислений

Несмотря на то, что в языке логического программирования отсутствуют традиционные управляющие конструкции, такие как оператор перехода, условный оператор, оператор цикла, программисту доступны механизмы управления процессом перебора всех возможных состояний.

Возврат или откат. Откат — это попытка найти следующий вариант решения задачи. Откат вызывается неудачей в некотором месте программы, что приводит систему к попытке найти следующее решение. Откат идет до места, где возможно вычислить другое решение. При этом все конкретизированные переменные освобождаются.

Для такой организации вычислительного процесса служит предикат `fail`, который всегда завершается неуспешно. В этом

случае поиск всех подходящих решений возможен путем перебора всех альтернатив, поиск которых осуществляется при неудачной попытке применения текущей.

Пусть в примере, иллюстрирующем работу процедуры «Вычислить», необходимо получить с помощью внутренней цели список всех больших животных. Для решения такой задачи можно ввести предикат:

```
show :- big(X), write(X), nl, fail.
```

Тогда внутренняя цель может быть сформулирована так:

```
goal
  show.
```

После нахождения первого решения («медведь») и вывода его на экран продолжится поиск других альтернатив («слон»), так как предикат `show` принудительно закончится неудачей.

Откат до ближайшей альтернативы происходит всегда в случае неуспешного завершения предиката, но только наличие предиката `fail` (или другого тождественно ложного) обеспечит перебор всех возможных альтернатив.

В следующем примере предикат-факт `country` содержит сведения о названии страны и населении. Программа выдает список стран, население которых превышает `1e7` (10^7 человек).

```
Predicates
    country(symbol,real)
    print_countries
Clauses
    country(england,3e7).
    country(france,2.3e7).
    country(germany,1.6e7).
    country(denmark,2.4e6).
    country(canada,7.3e6).
    country(chile,2.5e).
    print_countries :-
        country(X,P), %  рассматривается очередной факт
        P > 1e7,
        write(X), % вывод на экран названия страны
        nl,       % перевод строки
        fail.     % fail - встроенный предикат, означает "ложь"
    print_countries.
Goal  print_countries.
```

После того как выведен на экран очередной результат, стандартный предикат `fail` завершает работу предиката `print_countries` ложно и инициирует поиск других решений. Когда будут найдены все решения, второй предикат `print_countries` позволит программе завершиться истинно.

Механизм отсечения (ограничение перебора). В процессе достижения цели Prolog-система автоматически перебирает последовательно все варианты, осуществляя возврат при неуспехе какого-либо из них. Чтобы исключить в ряде случаев неэффективный ничем не ограниченный перебор, предусмотрен механизм отсечения.

Рассмотрим следующий пример. Пусть задана функция

$$y = f(x) = \begin{cases} 0, & x < 3; \\ 2, & 3 \le x < 6; \\ 4, & x \ge 6. \end{cases}$$

На языке Prolog такую функциональную зависимость можно выразить с помощью бинарного отношения `F(X,Y)`:

```
F(X,0) :- X<3.        % Правило 1
F(X,2) :- X>=3, X<6.  % Правило 2
F(X,4) :- X>=6.       % Правило 3
```

Предположим, что задана цель:

```
Goal F(1,Y), Y>2.
```

При вычислении первого правила Y конкретизируется 0, поэтому вторая цель принимает вид $0 > 2$ и терпит неудачу, а следом, очевидно, терпит неудачу и весь список целей. Однако, прежде чем сформировать результат, Prolog-система с помощью возврата пытается проверить применимость еще двух правил, хотя все три правила являются взаимоисключающими (успех возможен только в одном из них и нет смысла при этом проверять остальные).

Отсечение в Prolog запрещает для предиката механизм отката. Перепишем фрагмент программы с использованием отсечения (обозначается знаком «!»):

```
F(X,0) :- X<3, !.       % Правило 1
F(X,2) :- X>=3, X<6, !. % Правило 2
F(X,4) :- X>=6.         % Правило 3
```

Теперь на запрос цели

```
Goal F(1,Y), Y>2
```

будет выдан отрицательный ответ сразу после проверки правила 1 и конкретизации $Y = 0$, так как возврат в точку, помеченную символом «!», невозможен, что запрещает последующий откат и поиск других вариантов ответа.

Реализуем процедуру определения максимального из двух чисел с использованием механизма отката. Такую процедуру можно запрограммировать в виде отношения

```
Maximum(X, Y, Max)
```

где `Max = X`, если `X >= Y`, и `Max = Y`, если `X < Y`. Это соответствует двум правилам:

```
Maximum(X, Y, X) :- X>=Y.
Maximum(X, Y, Y) :- X<Y.
```

Эти правила являются взаимоисключающими. Если выполняется первое, второе потерпит неудачу. Если терпит неудачу первое, второе обязательно должно выполниться. Поэтому возможна формулировка, аналогичная понятию «иначе»:

```
Maximum(X, Y, X) :- X>=Y, !.
Maximum(X, Y, Y).
```

Программирование циклов (итерации). На языке Prolog могут быть реализованы алгоритмические методы, позволяющие добиться того же эффекта, который дает применение итеративных циклов в процедурных языках. Цикл можно реализовать с завершением или потенциально бесконечным.

Примеры циклов можно продемонстрировать с использованием предиката `repeat`, который ведет себя так, как если бы был запрограммирован следующим образом:

```
repeat.
repeat:-repeat.
```

`Repeat` — это цель, которая всегда успешна. Она недетерминирована, поэтому всякий раз, когда до нее доходит перебор, порождается новая ветвь вычислений.

Пример цикла с сочетанием `repeat` и `fail`.

```
run:-  repeat,
readln(X),
process(X),
write(X),
fail.
```

Предикат `fail` заставляет систему осуществлять возврат к `repeat` и `process` в случае успешного завершения.

С использованием предиката `repeat` можно также организовать любое подходящее условие окончания работы цикла:

```
run:-  repeat,
process,
write("Продолжать ввод? (y/n)" ),
readchar(Key), Key = 'n'.
```

Рекурсия. Правило является *рекурсивным*, если оно вызывает само себя в виде подцели, содержащейся в теле по крайней мере одного из его утверждений.

Примером рекурсивных вычислений является известный алгоритм вычисления факториала. В синтаксисе языка Prolog эта программа может иметь такой вид:

```
Domains
        N, F=real
Predicates
        factorial(N,F)
Clauses
        factorial(0,1): - !. % база рекурсии, ограничение
                             % вычислений
        factorial(N,R):- N1=N-1,
                  factorial(N1,R1),
R=R1*N.
Goal
        factorial(5,F),write("5! = ", F).
```

При каждом вызове предложения `factorial` генерируются новые переменные, которые действуют всегда только на своем уровне вложенности, пока не встретится условие прекращения вычислений. Только после этого на обратном пути прохождения рекурсии определяются результаты, которые передаются вверх.

Любое рекурсивное определение содержит, по крайней мере, одно нерекурсивное правило и одно или несколько правил с ре-

курсией. В большинстве случаев имеется по одному правилу каждого типа. Считается, что используется хвостовая рекурсия, если последнее условие в последнем правиле является рекурсивным. Такая рекурсия имеет преимущество перед нехвостовой рекурсией, так как позволяет ограничить рост стека и строго контролировать процесс возврата. Это происходит благодаря очистке стека после успешного сопоставления условия, содержащего рекурсию.

Рекурсию можно использовать для программирования цикла со счетчиком. В следующем примере цикл выполнится 10 раз.

```
for(10).
for(N):- N<10,
     readln(X),
     process(X,Y),
     write(Y),
     N1=N+1,
     for(N1).
Goal for(0).
```

В следующем примере продемонстрирован прием программирования бесконечного рекурсивного цикла. Выполняется ввод и печать символов. Процесс повторяется до нажатия клавиши [ENTER], имеющей кодовое число 13.

```
typewr('\13').
typewr(Char):- write(Char),
          readchar(C),
          typewr(C).
Goal readchar(Char), typewr(Char).
```

Списки

Списки являются одной из основных структур данных в языке Prolog. Список — это последовательность, составленная из произвольного числа элементов, например [1, 5, 33, 28, 99]. Основным механизмом работы со списками является рекурсия, позволяющая компактно описывать алгоритмы. Для рекурсии не требуется знания точного числа элементов списка, достаточно определения, как правило, головного элемента списка и условия окончания работы рекурсии.

Все структурные объекты в языке Prolog являются деревьями. Чтобы представить список в виде такого стандартного объекта, введены два понятия: *голова* и *хвост*. Первый элемент

списка является его головой, а оставшиеся — его хвостом. Тогда список — это структура данных, определяемая следующим образом:

- список может быть пуст;
- список состоит из головы и хвоста, причем головой списка может быть элемент любого типа, а хвост должен быть списком.

Для обозначения списка используются квадратные скобки, а для отделения головы списка от его хвоста — символ "|". Пустой список обозначается, как [ ].

Например, для списка [1|5,33] элемент 1 является головой, а список [5,33] хвостом.

Перед использованием списков в программе необходимо сначала описать домен списка в разделе `domains`:

```
<имя_домена> = <тип_элемента_списка>*
```

Например, для списка с целочисленными элементами:

```
domains
   list_nmb = integer*
```

Рассмотрим некоторые операции над списками.

Проверка принадлежности элемента списку. Пусть требуется определить предикат:

```
member(X,L)
```

где `L` — список; `X` — объект того же типа, что и элементы списка `L`.

Цель должна быть истинной, если элемент `X` встречается в `L`. Запись предиката может быть основана на следующих альтернативных соображениях:

а) `X` есть голова списка `L`;

б) `X` принадлежит хвосту.

Это может быть записано в виде двух предложений, первое из которых есть простой факт, ограничивающий вычисления, а второе — рекурсивное правило:

```
member(X, [X| _ ]).
member(X, [ _ | T ]):- member(X,T).
```

Знаком подчеркивания обозначена анонимная переменная, значение которой в данном предложении неважно.

Если список L будет пуст, предикат завершится ложно, так как для пустого списка нет своего правила.

Сцепление (конкатенация) списков. Для сцепления списков определим предикат

```
concat (L1,L2,L3).
```

Здесь L1 и L2 — сцепляемые списки, а список L3 представляет собой результат конкатенации.

Для определения предиката выделим два случая (в зависимости от вида первого аргумента L1):

1. Если первый список пуст, то второй и третий представляют собой один и тот же список:

```
Lconcat ([ ],L,L).
```

2. Если первый список не пуст, то он имеет голову и хвост: [X|L1]. Его сцеплением со списком L2 будет список [X|L3], где список L3 получен после сцепления списков L1 и L2, т. е.

```
concat ([X|L1],L2,[X|L3]) :- concat (L1,L2,L3).
```

Пусть на входе — списки L1 = [a,b,c], L2 = [1,2]. Предикат имеет вид:

```
concat([a,b,c], [1,2],[])
```

Выполняя второе (рекурсивное) правило, программа будет выталкивать поочередно элементы первого списка в стек до тех пор, пока первый список не станет пустым. Далее выполняется первое правило и предикат примет вид (третий список инициализируется вторым):

```
concat([], [1,2],[1,2])
```

После этого происходит возврат элементов из стека и приписывание их в начало третьего списка (в соответствии со вторым правилом). В результате получаем:

```
L3 = [a,b,c,1,2]
```

Добавление элемента в список. Наиболее простой способ добавить элемент в список — это вставить его в самое начало так,

чтобы он стал новой головой. Если X — это новый элемент, который добавляют в список L, то результирующий список будет выглядеть так:

```
[X|L]
```

То есть, чтобы добавить элемент в список, не надо использовать предикат. Однако, если стоит задача определить такой предикат, то это можно сделать следующим образом:

```
add(X,L,[X|L])
```

Предикат имеет три аргумента — добавляемый элемент, список и результирующий список.

Удаление элемента из списка. Удаление элемента X из списка L можно описать в виде отношения

```
del(X,L,L1),
```

где L1 — это список L, из которого удален элемент X.

Операцию удаления можно сформулировать следующим рекурсивным образом:

а) если X является головой списка, тогда результатом удаления будет хвост этого списка:

```
del (X, [X|T],T).
```

б) если X находится в хвосте списка, тогда его нужно удалить из хвоста:

```
del (X, [Y|T], [Y|T1]):- del (X,T,T1).
```

Если элемент X есть голова списка, то выполняется первое правило. Если элемент расположен в хвосте — работает рекурсивное правило, пересылающее в стек поочередно элементы списка до тех пор, пока очередным элементом не станет X. Тогда выполнится первое правило, хвост списка проинициализирует результирующий список и последовательно осуществится возврат элементов из стека в соответствии с рекурсивным правилом:

```
?- del (a, [c, a, b, 1, 2],T).
T = [c, b, 1, 2]
```

Попытка удалить элемент, не содержащийся в списке, потерпит неудачу.

Удаление из списка повторяющихся элементов. Аргументами предиката `unikue` являются два списка — исходный и результирующий. Смысл алгоритма прост: если элемент присутствует в списке (проверяется предикатом `member`), то он не записывается в результирующий список, иначе добавляется в качестве головного элемента.

```
unikue([ ],[ ]).
unikue([H|T], L):- member(H,T), unikue(T,L).
unikue([H|T], [H|L]):- unikue(T,L).
```

Выделение подсписка. Предикат выделения подсписка имеет два аргумента — список `L` и список `S` такой, что `S` содержится в `L`. Предикат можно сформулировать следующим образом: `S` является подсписком `L`, если:

а) `L` можно разбить на 2 списка `L1` и `L2`;

б) `L2` можно разбить на 2 списка `S` и `L3`.

Для разбиения списков можно использовать предикат `concat`, определенный ранее, поэтому запись предиката `sublist` — выделение подсписка — может быть следующей:

```
sublist(S, L) : - Lconcat(L1,L2,L), Lconcat(S, L3, L2).
```

Например, предикат `sublist ([2,3], [1,2,3,4])` успешен, так как успешны предикаты `concat ([1], [2,3,4], [1,2,3,4])` и `concat ([2,3], [4], [2,3,4])`.

Обращение списка. Определим предикат

```
reverse(L1,L2).
```

Аргументы `L1` и `L2` — два списка, из которых список `L2` содержит элементы списка `L1`, записанные в обратном порядке.

```
reverse(L1,L2):-reversel(L1,[],L2).
reversel([],L,L).
reversel([H|T],L1,L2):-reversel(T,[H|L1],L2).
```

Предикат `reverse` в данном случае является интерфейсным, он запускает в работу основной рабочий предикат `reversel`, имеющий дополнительный второй аргумент — список, который

вначале пуст, и используется собственно для обращения списка. На каждом шаге рекурсии один элемент исходного списка становится головой промежуточного списка. Третий аргумент передается от шага к шагу и конкретизируется в момент достижения базового состояния предиката `reverse1`. Когда первый список исчерпан, второй уже содержит элементы, записанные в обратном порядке. Отметим, что наличие третьего аргумента, фиксирующего результат, обязательно, так как после обратного прохождения рекурсии все конкретизированные переменные принимают свои первоначальные значения.

Упражнения

1. На языке Java опишите класс, расширяющий класс `StringBuffer` следующими процедурами и функциями обработки строк:

- посчитать количество слов в строке (разделителем считать символ пробела);
- посчитать количество предложений в строке (разделителями считать точку, вопросительный и восклицательный знаки);
- посчитать количество гласных (согласных) букв в строке;
- посчитать количество знаков препинания (знаки препинания — запятая, точка, точка с запятой, двоеточие);
- удалить из строки повторы слов;
- определить, можно ли из последних трех слов 1-й фразы составить три первых слова 4-й фразы (разделителем слов считать символ пробела, а разделителем фраз — точку с пробелом);
- переписать строку в обратном порядке.

2. Программа на языке Prolog представляет собой последовательность фактов:

```
father_or_mother (анна, николай). /* Анна - родитель Николая */
father_or_mother (иван, николай).
father_or_mother (иван, екатерина).
father_or_mother (николай, анастасия).
father_or_mother (николай, полина).
father_or_mother (полина, дмитрий).
```

Каковы будут ответы на следующие запросы:

а) `? - father_or_mother (дмитрий, X).`

б) `? - father_or_mother (X, иван).`

в) `? - father_or_mother (анна, X), father_or_mother (X, полина).`

г) `? - father_or_mother (анна, X), father_or_mother (X, Y), father_or_mother (Y, дмитрий).`

3. Сформулируйте следующие запросы к программе упражнения 4:

а) Кто родитель Екатерины?

б) Имеет ли Дмитрий ребенка?

в) Кто является родителем родителя Полины?

4. Определите предикат `внук`, используя предикат `родитель (father_or_mother)` упражнения 3.

5. Запишите на языке Prolog следующие правила:

а) Всякий, кто имеет родителя, является ребенком.

б) *X* является сестрой *Y*, если у *X* и *Y* есть общий родитель и *X* — женщина.

в) Всякий, кто имеет ребенка, счастлив.

г) Всякий *X*, имеющий ребенка, у которого есть сестра, имеет двух детей.

6. Дана программа на языке Prolog:

```
recurs (1, one).
recurs (r(1), two).
recurs(r(r(1)), three).
recurs(r(r(r(X))), Y) :- recurs(X, Y).
```

Напишите ответы на следующие запросы:

а) `? - recurs(r(1), A).`

б) `? - recurs(r(r(1)), one).`

в) `? - recurs(B, three).`

г) `? - recurs(r(r(r(r(r(r(1)))))), C).`

7. Перепишите следующую программу на языке Prolog, не используя точку с запятой («;»):

```
change (Number, Word) :- Number = 1, Word = один;
Number = 2, Word = два;
Number = 3, Word = три.
```

8. На языке Prolog задайте последовательность целей для порождения списка L2, получающегося из списка L вычеркиванием его трех первых и трех последних элементов.

9. На языке РЕФАЛ запишите образцы, которые соответствуют следующим словесным описаниям[1]:

а) выражение, оканчивающееся двумя одинаковыми символами;

б) выражение, которое содержит по крайней мере два одинаковых терма на верхнем уровне структуры;

в) непустое выражение.

10. Вычислить результаты следующих сопоставлений на языке РЕФАЛ:

а) `'abbab' : e.1 t.X t.X e.2`

б) `'ab(b)ab' : e.1 t.X t.X e.2`

в) `A(B) C D (A (B)) : e.1 e.2 (e.1)`

г) `'160' : 16 e.X`

11. В рекурсивной арифметике натуральные числа представляются как 0, 0′, 0″ и т. д. Операция сложения задается соотношениями:

$x + 0 = x;$

$x + y' = (x + y)'.$

Определите на языке РЕФАЛ функцию `<Addrecurs (e.1)e.2>`, реализующую такую операцию сложения, используя символ `'0'` в качестве 0 и `'1'` вместо символа «′».

12. На языке РЕФАЛ определите функцию `<Addbinary (e.1) (e.2)>`, которая складывает числа в двоичном представлении.

13. На языке РЕФАЛ определите функцию `Less (<)` для целых чисел.

14. *Инвертированная* пара в последовательности чисел — это такая пара рядом стоящих чисел, в которой первое число больше второго. На языке РЕФАЛ задать следующие функции:

1) найти в заданной числовой последовательности первую инвертированную пару, для которой сумма членов пары превосходит 100;

---

[1] Ответы к упражнениям для языка РЕФАЛ приведены в Приложении 2.

2) определить, превосходит ли сумма членов первой инвертированной пары число 100.

15. Функция *N*! (факториал) может быть вычислена перемножением чисел натурального ряда от 1 до *N*. Определить РЕФАЛ-функцию, которая использует этот алгоритм.

16. На языке РЕФАЛ определите функцию `Reverse`, которая переписывает строку в обратном порядке.

17. Конечное множество символов может быть представлено строками, включающими те и только те символы, которые входят в это множество. На языках Java и РЕФАЛ определите следующие функции:

а) вычислить пересечение двух множеств;

б) вычислить объединение двух множеств.

18. Палиндром — строка, одинаково читаемая как слева направо, так и справа налево. На языке Prolog задать предикат `pal`, определяющий, является ли строка палиндромом. Определить аналогичную функцию на языках Java и РЕФАЛ.

Контрольные вопросы

1. Охарактеризуйте методологию императивного программирования.
2. Дайте определение машины Тьюринга.
3. Перечислите типовые операторы, характерные для процедурных языков программирования.
4. Сформулируйте понятие объекта.
5. Перечислите и охарактеризуйте основные свойства объектно-ориентированного программирования.
6. Сформулируйте различия между компонентом и объектом.
7. Охарактеризуйте визуальные и невизуальные компоненты.
8. Перечислите и охарактеризуйте стандартные типы данных языка Java.
9. Приведите определение класса для языка Java.
10. Приведите определение метода и охарактеризуйте спецификаторы метода для языка Java.
11. Дайте определение метода-конструктора.
12. Охарактеризуйте классы `String` и `StringBuffer`.
13. Дайте определение алгоритма Маркова.
14. Охарактеризуйте методологию функционального программирования.

15. Приведите определение функции языка РЕФАЛ.
16. Сформулируйте общий алгоритм работы РЕФАЛ-машины.
17. Охарактеризуйте методологию логического программирования.
18. Дайте определение дизъюнкта Хорна.
19. Охарактеризуйте понятия «унификация» и «резолюция».
20. Приведите определение структуры языка Prolog.
21. Дайте определение предиката, факта, правила, цели для языка Prolog.
22. Охарактеризуйте структуру программы на языке Prolog.
23. Дайте определение списка языка Prolog.

Глава 3
ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММ

Во второй главе были рассмотрены методики программирования, которые дают разный выигрыш при решении задач различных классов. Этот выигрыш можно оценивать, например, по таким параметрам, как:

- эффективность программного обеспечения на современных ЭВМ;
- общие затраты на разработку программного обеспечения.

Со временем одни языки, поддерживающие определенные методологии программирования, развивались, приобретали новые черты и до сих пор остаются востребованными, другие утратили свою актуальность и сегодня представляют в лучшем случае чисто теоретический интерес. В значительной степени это связано с такими факторами, как:

- наличие инструментальной системы, поддерживающей разработку приложений на конкретном языке программирования;
- удобство разработки, сопровождения и тестирования программ;
- стоимость разработки с применением конкретного языка программирования.

Инструментальной системой будем называть функционирующую под единой оболочкой некоторую логически связанную совокупность программных инструментов, поддерживающую процессы разработки и сопровождения программных средств (ПС) на определенном языке программирования.

В идеальном варианте инструментальные системы должны распространяться на максимально возможное количество процессов и покрывать максимум стадий жизненного цикла ПС. На

самом деле в современной практике программирования сложилась следующая ситуация:

- процессы программирования, тестирования и отладки ПС в основном поддерживают системы и инструментальные среды программирования;
- анализ и проектирование ПС обеспечивают средства автоматизации разработки программ.

Далее будут рассмотрены этапы жизненного цикла программных систем, вопросы обеспечения качества и надежности, инструментальные среды разработки программ и CASE-средства, а также методы и языки моделирования программных систем.

3.1. Жизненный цикл программных средств

Жизненный цикл (ЖЦ) программных средств отражает весь процесс их создания и применения. Этот процесс состоит из нескольких взаимосвязанных этапов, выполняющихся в определенной последовательности и обеспечивающих ведение разработки ПС на всех стадиях — от подготовки технического задания до завершения.

Модели жизненного цикла ПС несколько различаются терминологией и графическим представлением этапов и их взаимодействия. Практически во всех моделях отражен ряд базовых этапов, к которым относятся:

- системный анализ и первоначальная разработка общих требований к ПС;
- структурное проектирование, разработка спецификаций предварительного проекта ПС;
- разработка программных компонентов и их комплексирование;
- комплексная отладка, испытание и сертификация ПС;
- сопровождение, модернизация и развитие версий ПС;
- документирование разработки и результирующих продуктов ПС.

Первый этап — системный анализ и определение требований — устанавливает общие требования к ПС по надежности, технологичности, правильности, универсальности, эффективности, информационной согласованности и т. д. Они дополняются требованиями заказчика, включающими необходимые функции и воз-

можности, пространственно-временные ограничения, режимы функционирования, требования точности и надежности и т. д., т. е. вырабатывается описание системы с точки зрения пользователя. При определении спецификаций проводится точное описание функций ПС; разрабатываются и утверждаются входные и промежуточные языки, форма входной и выходной информации; описывается возможное взаимодействие с другими программными комплексами; специфицируются средства расширения и модификации ПС; разрабатываются интерфейсы; решаются вопросы информационного обеспечения; утверждаются основные алгоритмы.

Итогом выполнения этого этапа являются эксплуатационные и функциональные спецификации, содержащие конкретное описание ПС. Разработка спецификаций требует тщательной работы системных аналитиков в постоянном контакте с заказчиками, которые в большинстве случаев не способны сформулировать четкие и реальные требования.

Эксплуатационные спецификации содержат сведения о быстродействии ПС, затратах памяти, требуемых технических средствах, надежности и т. д.

Функциональные спецификации определяют функции, которые должно выполнять ПС, т. е. в них определяется, *что* надо делать, а не то, *как* это делать.

Спецификации должны быть полными, точными и ясными. Полнота исключает необходимость получения разработчиками ПС в процессе их работы от заказчиков иных сведений, кроме тех, которые содержатся в спецификациях. Точность не позволяет вводить различные толкования содержания, а ясность предполагает легкость понимания как заказчиком, так и разработчиком.

Назначение спецификаций:

- спецификации являются заданием на разработку ПС и их выполнение — закон для разработчика;
- спецификации используются для проверки готовности ПС;
- спецификации составляют неотъемлемую часть программной документации, облегчают сопровождение и модификацию ПС.

Второй этап — структурное проектирование ПС. На этом этапе:

- создается представление архитектуры ПС;
- устанавливается состав и структура модулей;

- разрабатываются алгоритмы, задаваемые спецификациями;
- выбирается структура информационных массивов;
- фиксируются межмодульные интерфейсы, входные и выходные формы данных.

Цель этапа — декомпозиция сложных задач создания ПС на подзадачи меньшей сложности. Результатом работы на этом этапе являются спецификации на отдельные модули, дальнейшая декомпозиция которых нецелесообразна.

Третий этап — программирование (кодирование). На данном этапе проводится программирование модулей. Этап считается менее сложным по сравнению со всеми остальными. Проектные решения, полученные на предыдущем этапе, реализуются в виде программ. Разрабатываются и подключаются отдельные функциональные блоки. Одна из задач этапа — обоснованный выбор языков программирования. На этом же этапе решаются все вопросы, связанные с особенностями типа ЭВМ.

Четвертый этап — тестирование и отладка ПС. Этап предполагает проверку работоспособности ПС, а также его соответствие спецификациям. Тестирование определяется, как любая деятельность, выполняемая с целью обнаружения ошибок, а отладка — как процесс их локализации и устранения. Каждый тест (или тестовый вариант) определяет, во-первых, набор исходных данных и условий для запуска программы, во-вторых — набор ожидаемых результатов работы программы. Проверка выполнения всех требований, всех структурных элементов ПС проводится на таком количестве тестов, которое удовлетворяет определенным требованиям полноты тестирования.

Пятый этап — эксплуатация и сопровождение, т. е. процесс исправления выявленных в результате эксплуатации ошибок, координации всех элементов системы в соответствии с требованиями пользователя, внесения (в соответствии с требованиями пользователей) необходимых исправлений и изменений. Выделение такого этапа вызвано как минимум двумя причинами: во-первых, в ПС остаются ошибки, не выявленные при отладке; во-вторых, пользователи в ходе эксплуатации ПС настаивают на внесении в него изменений и его дальнейшем совершенствовании.

Следует отметить, что сопровождение ПС предполагает повторное применение каждого из предшествующих этапов ЖЦ к существующей программе, но не разработку новой программы.

Модели жизненного цикла

Повышение эффективности разработки ПС в целом достигается за счет: регламентации порядка проведения работ, автоматизации этапов и операций поддержки жизненного цикла, разделения труда между специалистами разной квалификации и проблемной ориентации применяемой технологии.

Наибольшее число современных моделей ЖЦ ориентировано на сложные, критические ПС обработки информации и управления в реальном времени. К таким ПС предъявляются наиболее высокие требования к качеству функционирования, они создаются коллективами специалистов в течение длительного времени. С другой стороны, для ПС этого класса характерны использование типовых систем управления данными, относительно небольшой объем оригинальных программных разработок и широкое применение повторно используемых компонентов. Такая специфика отражается в модели ЖЦ. Например, ЖЦ информационных систем обработки больших потоков данных в сфере организации и управления увеличивает затраты на проектирование и сокращает затраты на разработку оригинальных программных компонентов.

С точки зрения использования модели ЖЦ рассматривают следующие стратегии конструирования ПС [27]:

- *однократный подход* — линейная последовательность этапов конструирования;
- *инкрементная стратегия*, согласно которой в начале процесса определяются все системные и пользовательские требования, а оставшиеся этапы ЖЦ выполняются в виде последовательности версий: первая версия реализует некоторую часть запланированных возможностей, а каждая следующая версия дополняет возможности предыдущей до тех пор, пока не будут удовлетворены все требования;
- *эволюционная стратегия*, предполагающая итерационное построение конкретного программного продукта, но (в отличие от инкрементной стратегии) в начале разработки определены не все требования (они уточняются в результате разработки версий).

Охарактеризуем некоторые модели, поддерживающие перечисленные стратегии.

Каскадная модель. Обеспечивает стратегию однократного подхода и представляет собой классическую модель ЖЦ, разработанную Уинстоном Ройсом в 1970 г.

Разработка ПС рассматривается как линейная последовательность этапов, причем переход на следующий (нижний) этап происходит только после завершения работ на текущем этапе (рис. 3.1).

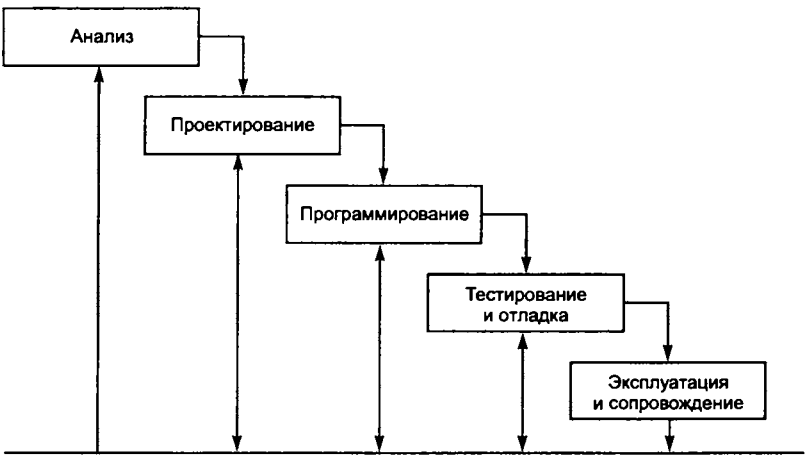


Рис. 3.1. Каскадная модель ЖЦ

К достоинствам такой модели относят определение четкого плана и временного графика работ по всем этапам ЖЦ и упорядочивание хода разработки.

Недостатками модели являются:

- невозможность отклонения от стандартной строго определенной последовательности этапов (чего часто требуют реальные проекты);
- исходные требования к разработке должны быть сформулированы точно в начале цикла работ, хотя чаще всего в начале проекта требования заказчика определены лишь частично;
- результаты работы над проектом доступны заказчику только в конце работы.

Инкрементная модель. Является классическим примером применения инкрементной стратегии и представляет собой повторяющийся цикл линейных последовательностей каскадной модели.

Каждая отдельная линейная последовательность вырабатывает очередной инкремент ПС, который может быть поставлен за-

казчику. Первый инкремент приводит к формированию базового продукта, реализующего базовые требования (при этом многие вспомогательные функции остаются нереализованными). План следующего инкремента предусматривает модификацию базового программного продукта, обеспечивающую ряд дополнительных функций и т. д. Тем самым инкрементная модель обеспечивает после выполнения каждого отдельного инкремента работающий продукт (рис. 3.2).

Рис. 3.2. Инкрементная модель ЖЦ

Современная реализация инкрементного подхода — *экстремальное программирование* (XP), предложенное Кентом Беком (1999 г.). XP-процесс ориентирован на очень малые приращения функциональности при условии неопределенных или быстро меняющихся требований. Четырьмя базовыми действиями в линейной последовательности XP-цикла являются: проектирование, программирование, тестирование, работа с заказчиком. Одной из основных характеристик цикла является непрерывная связь с заказчиком.

Быстрая разработка приложений (Rapid Application Development — RAD) также на сегодняшний день является распространенной реализацией инкрементной стратегии и обеспечивает экстремально короткий цикл разработки ПС за счет использования компонентно-ориентированного конструирования (рис. 3.3).

RAD-подход ориентирован на разработку информационных систем. Линейная последовательность цикла включает следующие этапы:

- *бизнес-моделирование* (определение информационных потоков между бизнес-функциями);

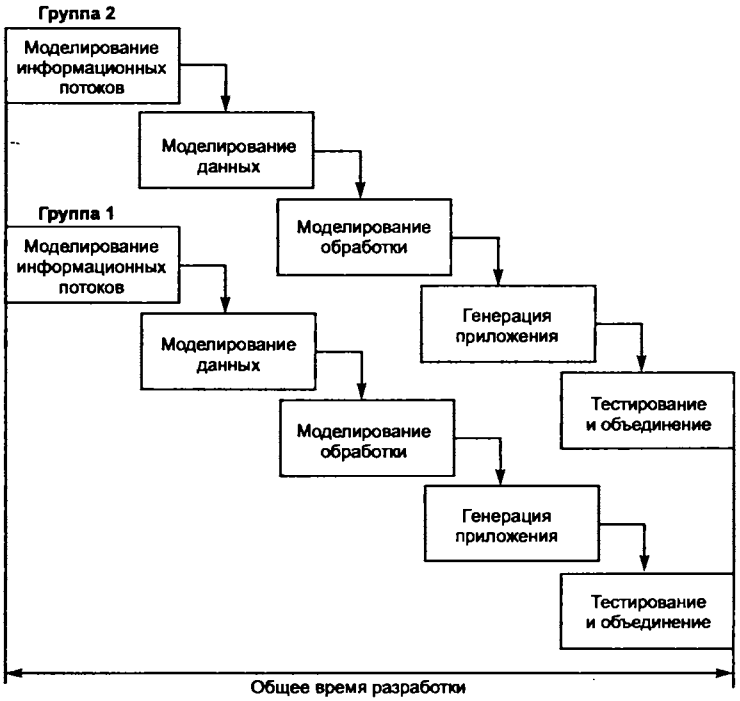


Рис. 3.3. Модель быстрой разработки приложений

- *моделирование данных* (построение объектной модели информационных потоков — выделение объектов, их свойств и связей);
- *моделирование обработки* (описание процессов преобразования данных, обеспечивающих бизнес-функции, например, добавление, модификацию, поиск, удаление);
- *генерация приложения* (работа в средах быстрой разработки приложений, повторное использование программных компонентов или создание повторно используемых компонентов);
- *тестирование и объединение* (общее время тестирования сокращается за счет повторного использования уже протестированных компонентов).

Применение RAD возможно в том случае, когда разработка и реализация главных функций продукта может быть поделена между отдельными группами разработчиков (каждой отдельной группе адресуется своя функция).

Среди недостатков RAD можно отметить, что модель применима только в проектах, где можно провести декомпозицию приложения на отдельные модули без потери в производительности, а также тот факт, что реализация модели в случае крупных разработок требует существенных людских ресурсов.

Спиральная модель. Автор модели — Барри Боэм (1988 г.). Модель добавляет в процесс разработки новую стадию — анализ риска — и является классическим примером применения эволюционной стратегии разработки.

Внимание модели концентрируется на итерационном процессе начальных этапов проектирования. Всего модель определяет четыре этапа (рис. 3.4):

- планирование (определение целей, вариантов и ограничений);
- анализ риска (анализ вариантов и распознавание риска);
- конструирование (разработка проектов каждого нового уровня);
- оценивание (оценка заказчиком текущих результатов конструирования).

Продвижение по спирали определяет все более полные версии ПС. В первом витке спирали определяются начальные требования, анализируется риск. Если в результате анализа риска выявлена неопределенность требований, то этап конструирования обеспечивает только создание модели (или макета) продукта, который оценивает заказчик. Следующий виток планирования и анализ риска базируются на предложениях заказчика. Таким образом, углубляются и последовательно конкретизируются детали проекта ПС. Анализ риска в качестве результата форми-

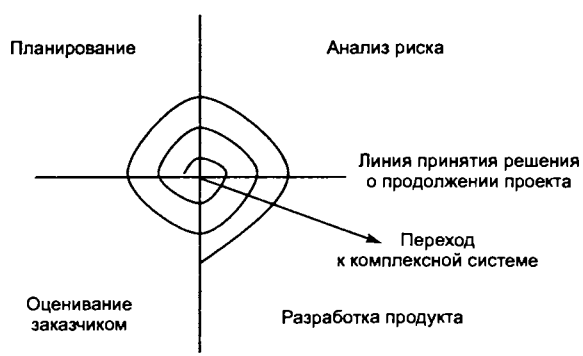


Рис. 3.4. Спиральная модель ЖЦ

рует рекомендации по продолжению проекта или прекращению работ (если риск слишком велик).

Следует отметить, что этап конструирования на каждом новом витке охватывает все больше этапов классического ЖЦ.

К достоинствам спиральной модели следует отнести

- наиболее реальное (эволюционное) отображение процесса разработки ПС;
- возможность явного учета риска на каждом витке спирали;
- использование моделирования и макетирования для уменьшения риска и совершенствования ПС.

Недостатки модели — повышенная требовательность к заказчику и трудности, связанные с контролем временных затрат и управлением разработкой.

Стандартизация этапов ЖЦ

В большинстве моделей ЖЦ ПС детализация ограничивается 8—10-ю крупными процессами или этапами. Для практического применения моделей при реальном планировании и управлении проектами необходима более подробная информация о содержании процессов. В подобных описаниях должны быть представлены исходные данные, содержание частных работ и ожидаемые результаты их выполнения, а также структура и содержание документов, сопутствующих их реализации.

Одним из путей повышения экономической эффективности создания ПС являются стандартизация и автоматизация технологических процессов проектирования, разработки и сопровождения программ для ЭВМ. В стандартах обобщаются опыт и результаты исследований множества специалистов, сосредотачивается методическая база. Это способствует повышению качества ПС и снижению затрат на их создание. Тем самым, к поддержке жизненного цикла ПС можно отнести стандарты, включающие языки программирования, интерфейсы с внешней средой, графику, а также непосредственно поддерживающие планирование и управление технологическими процессами проектирования, разработки и сопровождения ПС.

Стандарты могут использоваться как директивные, руководящие или как рекомендательные документы, а также как организационная база при создании средств автоматизации соответствующих технологических этапов или процессов. Стандартизация процессов отражается не только на их технико-экономических

показателях, но и, что особенно важно, на качестве создаваемых ПС. Качество программ тесно связано с методами и технологией их разработки, поэтому важной группой стандартов в этой области являются стандарты по обеспечению качества ПС.

Для проектирования ПС систем военного назначения министерством обороны США в 1988 г. был разработан и утвержден стандарт DOD-STD-2167A. Этим стандартом регламентированы 8 фаз (этапов) и около 30 обязательных отчетных документов в процессе создания критических ПС. Сформулировано около 250 типовых обязательных требований к процессам и объектам проектирования на всех этапах разработки.

В стандарте DOD представлена часть жизненного цикла ПС, отражающая только непосредственную разработку. Отсутствуют этапы эксплуатации и сопровождения, а также не полностью раскрыты процессы управления разработкой и интегральные процессы технологической поддержки жизненного цикла ПС.

Наиболее полно ЖЦ сложных программных средств регламентирован требованиями стандарта ISO 12207: 1995 «Информационные технологии. Процессы жизненного цикла программного обеспечения». Он определяет архитектуру, процессы, разделы и подразделы ЖЦ ПС, а также перечень базовых работ и детализацию содержания каждой из них. Архитектура ЖЦ ПС в стандарте базируется на трех крупных компонентах:

- основы жизненного цикла ПС и определяющие работы;
- процессы и работы, поддерживающие жизненный цикл ПС;
- организация и управление жизненным циклом ПС.

В табл. 3.1 приведен перечень некоторых стандартов обеспечения жизненного цикла ПС.

Таблица 3.1. **Стандарты жизненного цикла ПС**

Обозначение	Название стандарта
ISO/IEC 12207:1995	Информационные технологии. Процессы жизненного цикла программного обеспечения
ISO/IEC TR 15271:1998	Информационные технологии. Руководство по применению ISO/IEC 12207 (Процессы жизненного цикла программных средств)
ISO/IEC TR 15846:1998	Информационные технологии. Процессы жизненного цикла программного обеспечения. Управление конфигурацией
ISO 15226:1999	Техническая документация на продукцию. Модель жизненного цикла и назначение документов

Окончание табл. 3.1

Обозначение	Название стандарта
ISO/IEC 12207:1995/Amd.1:2002	Информационные технологии. Процессы жизненного цикла программного обеспечения. Изменение 1
ISO/IEC 12207:1995/Amd.2:2004	Информационные технологии. Процессы жизненного цикла программного обеспечения. Изменение 2
ISO/IEC 16085:2004	Информационные технологии. Процессы жизненного цикла программного обеспечения. Управление рисками
ISO/IEC 15289:2006	Разработка систем и программного обеспечения. Содержимое систем и информационные продукты обработки жизненного цикла программного обеспечения (Документация)
ISO/IEC 23026:2006	Разработка программного обеспечения. Рекомендуемая практика для Интернета. Разработка веб-сайтов, администрирование веб-сайтов и жизненный цикл веб-сайтов
ISO/IEC 14764:2006	Разработка программного обеспечения. Процессы жизненного цикла программного обеспечения. Сопровождение

3.2. Показатели качества и надежности программных средств

Формализация показателей качества в стандартах

Реализация технической политики в области обеспечения качества функционирования ПС базируется на использовании системы требований, нормативных документов и методик проверки степени их выполнения.

Формализация показателей качества программных средств отражена в группе соответствующих нормативных документов. В международных стандартах, посвященных оценке качества программных продуктов, при отборе минимума стандартизируемых показателей выдвигались и учитывались следующие принципы:

- ясность и измеряемость значений;
- отсутствие перекрытия между используемыми показателями;
- соответствие установившимся понятиям и терминологии;
- возможность последующего уточнения и детализации.

В стандартах выделены характеристики, которые позволяют оценивать ПС с позиции пользователя, разработчика и управляющего проектом. Все характеристики определены очень крат-

ко, без комментариев и рекомендаций по их применению к конкретным системам и проектам.

К основным характеристикам качества ПС отнесены:

1. Функциональная пригодность.
2. Надежность.
3. Применимость.
4. Эффективность.
5. Сопровождаемость.
6. Переносимость.

Каждая из характеристик имеет свои черты на следующем уровне детализации:

Функциональная пригодность:

1) пригодность для применения;
2) точность;
3) защищенность;
4) способность к взаимодействию и согласованность со стандартами и правилами проектирования.

Надежность:

1) уровень завершенности (отсутствие ошибок);
2) устойчивость к ошибкам и перезапускаемость.

Применимость:

1) понятность;
2) обучаемость;
3) простота использования.

Эффективность:

1) ресурсная экономичность;
2) временная экономичность.

Сопровождаемость:

1) удобство для анализа;
2) изменяемость;
3) стабильность;
4) тестируемость.

Переносимость:

1) адаптируемость;
2) структурированность;
3) замещаемость;
4) внедряемость.

Близким к международным стандартам по идеологии, структуре и содержанию является ГОСТ 28195—89 «Оценка качества программных средств. Общие положения». На верхнем, самом первом уровне в нем выделено шесть показателей — факторов качества: *корректность, надежность, удобство применения, эффективность, сопровождаемость* и *универсальность*. Эти факторы на втором уровне детализируются в совокупности 19 критериями качества. Состав используемых факторов предлагается выбирать в зависимости от назначения, функций и этапов жизненного цикла ПС.

В перечисленных документах представлен широкий спектр показателей и общее описание их содержания. Однако материалы имеют справочный характер и не содержат рекомендаций по выбору и упорядочению необходимого минимума критериев в зависимости от особенностей объекта и среды разработки. Кроме того, для большинства показателей отсутствуют методики их измерения и сопоставления, а также рекомендации, на каких этапах разработки их целесообразно применять.

Общие понятия программы, программного средства, программного продукта и их качества формализуются в стандарте ГОСТ 28806—90 «Качество программных средств. Термины и определения». Здесь даются определения наиболее употребляемых терминов, связанных с оценкой характеристик программ, уточнены понятия базовых показателей качества, приведенных в стандарте 28195—89.

Приведем некоторые определения.

Программное средство — программа, предназначенная для многократного применения на различных объектах разработчика любым способом и снабженная комплектом программных документов.

Программный продукт — набор компьютерных программ, процедур и связанная с ними документация и данные.

Функциональная пригодность — набор атрибутов, определяющий назначение, номенклатуру, необходимые и достаточные функции ПС, заданные техническим заданием (ТЗ) заказчика или потенциального пользователя.

В процессе проектирования ПС атрибуты функциональной пригодности конкретизируются и отражаются в соответствующих спецификациях на отдельные компоненты. Функциональную пригодность отражают также специализированные критерии, тесно связанные с конкретными функциями программ.

Корректность программы — характеристика ПС, которая определена только в области изменения исходных данных, заданных требованиями спецификации, и не зависит от динамики функционирования программы в реальном времени. Степень некорректности программ тем самым определяется вероятностью попадания реальных исходных данных в область, которая задана требованиями спецификации и технического задания, но не была проверена при тестировании и испытаниях.

Надежная программа — это программа, которая, прежде всего, должна обеспечивать достаточно низкую вероятность отказа в процессе функционирования в реальном масштабе времени.

Высокую надежность программ обеспечивают:

- быстрое реагирование на искажения программ, данных или вычислительного процесса;
- восстановление работоспособности за время, меньшее, чем порог между сбоем и отказом.

Работоспособность ПС можно гарантировать только при конкретных исходных данных, которые использовались при отладке и испытаниях. В реальных же условиях по различным причинам исходные данные могут попадать в области значений, не проверенные при испытаниях, а также не заданные требованиями спецификации, и вызывать сбои в работе ПС. При таких исходных данных функционирование программ трудно предсказать заранее, и весьма вероятны различные аномалии, завершающиеся отказами. Если при этом происходит достаточно быстрое восстановление, такое что не фиксируется отказ, то такие события не влияют на основные показатели надежности. Следовательно, надежность функционирования программ является понятием динамическим, проявляющимся во времени, и этим существенно отличается от понятия корректности программ.

Непредсказуемость вида, места и времени проявления дефектов ПС в процессе эксплуатации приводит к необходимости создания специальных, дополнительных *систем оперативной защиты* от непредумышленных, случайных искажений вычислительного процесса, программ и данных. Системы оперативной защиты предназначены для выявления и блокирования распространения негативных последствий проявления дефектов и уменьшения их влияния на надежность функционирования ПС до устранения их первичных источников. Для достижения такой цели в ПС должна вводиться временная, программная и информационная избыточность, осуществляющая оперативное обнаружение дефектов

функционирования, их идентификацию и автоматическое восстановление (рестарт) нормального функционирования ПС.

Первопричиной нарушения работоспособности программ — возникновения *сбоев* и *отказов* при безотказности аппаратуры — всегда является конфликт между реальными исходными данными, подлежащими обработке, и программой, осуществляющей эту обработку. Основным принципом классификации сбоев и отказов в программах при отсутствии их физического разрушения является *разделение по временному показателю длительности восстановления* после любого искажения программ, данных или вычислительного процесса, регистрируемого как нарушение работоспособности. Если длительность восстановления меньше заданного порога, то дефекты и аномалии, возникающие при функционировании программ, следует относить к *сбоям*. Если же длительность восстановления превышает пороговое значение — происходящее искажение соответствует *отказу*.

Качество и надежность функционирования ПС наиболее полно характеризуется следующими показателями:

- устойчивостью — способностью к безотказному функционированию;
- восстанавливаемостью работоспособного состояния после произошедших сбоев или отказов.

Устойчивость определяется как способность ПС реагировать на проявления ошибок так, чтобы это не отражалось на показателях надежности.

Восстанавливаемость характеризуется полнотой и длительностью процесса восстановления функционирования программ при перезапуске (рестарте). Перезапуск должен обеспечивать возобновление нормального функционирования ПС, на что требуются ресурсы ЭВМ и время, поэтому основным показателем процесса восстановления является *длительность восстановления*. Отказ ПС учитывает ситуации потери работоспособности, когда длительность восстановления достаточно велика и превышает пороговое значение времени, разделяющее события сбоя и отказа.

Процесс отладки программ происходит во времени, и его динамические характеристики могут служить частными конструктивными критериями для оценки достигнутого качества программ. Одним из таких критериев может служить *интенсивность обнаружения ошибок* или число ошибок, выявляемых в программе в процессе отладки за единицу времени (при постоянной интенсивности самого процесса отладки). В этом критерии отражаются

ошибки, приводящие к нарушению работоспособности программ, и дефекты, искажающие результаты, но не влияющие на надежность функционирования программ. В этом случае для интенсивности тестирования характерна положительная обратная связь — чем больше выявляется ошибок в программе на некотором интервале времени, тем шире должно быть варьирование тестовых данных и тестов. По мере устранения ошибок в программе частота их обнаружения снижается и специалисты, осуществляющие отладку, попадают в область низкого темпа обнаружения ошибок, при котором отладка завершается.

Методы обеспечения надежности функционирования программных средств

В общем случае под ошибкой в процессе создания или применения изделий или систем подразумевается дефект, погрешность или неумышленное искажение объекта или процесса. При этом предполагается, что известно правильное, эталонное состояние объекта, по отношению к которому может быть определено наличие отклонения — дефекта или ошибки. Для борьбы с ними необходимо исследовать факторы, влияющие на надежность ПС. Это позволит целенаправленно разрабатывать методы и средства обеспечения надежности сложных ПС различного назначения при реально достижимом снижении уровня дефектов проектирования.

При строго фиксированных исходных данных программы исполняются по определенным маршрутам и выдают совершенно определенные результаты. Многочисленные варианты исполнения программ при разнообразных исходных данных представляются для внешнего наблюдателя как случайные. В связи с этим дефекты функционирования программных средств, не имеющие злоумышленных источников, проявляются внешне как случайные, имеют разную природу и последствия. В частности, они могут приводить к последствиям, соответствующим нарушениям работоспособности, и к отказам при использовании ПС.

В [21] анализ надежности ПС базируется на модели взаимодействия основных компонентов (рис. 3.5). Модель рассматривает:

- *объекты уязвимости*, влияющие на надежность ПС;
- *дестабилизирующие факторы*, воздействующие на объекты уязвимости;
- *методы повышения надежности* ПС.

Рис. 3.5. Модель анализа надежности программных средств

Стандартизация качества и надежности ПС

В табл. 3.2 приведены некоторые международные стандарты в области обеспечения качества программных продуктов.

Таблица 3.2. **Стандарты, действующие в области обеспечения качества программных продуктов**

Обозначение	Название стандарта
ISO/IEC 9126-1:2001	Программирование. Качество продукта. Часть 1. Модель качества
ISO/IEC TR 9126-2:2003	Программирование. Качество продукта. Часть 2. Внешние показатели
ISO/IEC TR 9126-3:2003	Программирование. Качество продукта. Часть 3. Внутренние показатепи
ISO/IEC TR 9126-4:2004	Программирование. Качество продукта. Часть 4. Качество при использовании показателей
ISO/IEC 25051:2006	Технология программного обеспечения. Качество программного продукта. Требования и оценка. Требования к качеству коммерческого программного продукта и инструкции по испытанию

Порядок выбора и применения основных понятий качества, принципиальных концепций и критериев регламентируют международные стандарты в области *систем качества* (табл. 3.3).

Таблица 3.3. **Стандарты, действующие в области систем качества**

Обозначение	Название стандарта
ISO 9000:2005	Системы менеджмента качества. Основные положения и словарь
ISO 9001:2000	Системы менеджмента качества. Требования
ISO 9004:2000	Системы менеджмента качества. Рекомендации по улучшению деятельности
ISO 10007:2003	Системы менеджмента качества. Руководящие указания по менеджменту конфигурации
ISO 10005:2005	Системы менеджмента качества. Руководящие указания по планам качества
ISO/TR 10013:2001	Рекомендации по документированию систем менеджмента качества

В стандартах выделены три основные задачи в обеспечении качества:

- достижение и поддержка качества продукции или услуг на уровне, удовлетворяющем потребителей;
- обеспечение руководству уверенности в достижении и поддержке качества на заданном уровне;
- обеспечение потребителю уверенности в том, что заданное качество достигается или будет достигнуто.

Стандарты предназначены для унификации описания методов разработки и поставки ПС, а также способов контроля их качества, отвечающих требованиям заказчика. Необходимый уровень контроля качества может быть достигнут посредством предотвращения отклонений от стандартов на всех этапах ЖЦ от начала разработки до технического обслуживания. При этом предполагается, что контрактом на разработку ПС особо оговариваются важнейшие компоненты технологии проектирования и требования к техническим характеристикам ПС. Руководство компании-поставщика должно документально оформить цели, технологию и свои обязательства по обеспечению качества ПС. Должны быть определены ответственность, полномочия и взаимодействие всего персонала (руководящего, исполняющего ра-

боты и контролирующего), который влияет на качество создаваемого комплекса программ.

Действия по обеспечению и проверке качества должны проводиться персоналом поставщика, независимым от специалистов, непосредственно ответственных за выполнение работ и создание изделий. Они должны включать:

- анализ проекта;
- проверку системы обеспечения качества;
- контроль и испытания ПС при проектировании, производстве и обслуживании под управлением ответственного представителя руководства поставщика.

Система обеспечения качества выполняет в жизненном цикле ПС следующие функции:

- анализ содержания контракта, поддержанный методиками, обеспечивающими качество ПС;
- специфицирование требований заказчика, включая все функциональные и технические характеристики, необходимые для удовлетворения запросов заказчика;
- планирование процесса разработки: формализация этапов, график, ресурсы, методы и средства разработки, а также контроль и способы проверки результатов по этапам работ;
- планирование обеспечения качества компонентов, а также ПС в целом (должно актуализироваться и конкретизироваться по мере проведения разработки);
- определение методологии и средств выполнения работ по проектированию и реализации проекта, а также анализ их результатов с точки зрения обеспечения выполнения требований заказчика;
- планирование, реализацию, оценку результатов и документирование испытаний и сертификации;
- приемку заказчиком для завершения контракта по разработке или обслуживанию ПС.

3.3. Инструментальные среды разработки программ

С развитием языков и технологий программирования совершенствовались и средства разработки программ — от режима командной строки до интегрированной среды разработки приложений.

Системы программирования

Система программирования (СП) представляет собой совокупность средств разработки программ, обеспечивающих автоматизацию составления и отладки программ на конкретном языке программирования.

Современные системы программирования обычно предоставляют пользователям такие средства разработки программ, как:

- создание и редактирование текстов программ;
- компилятор или интерпретатор;
- библиотеки стандартных процедур и функций;
- утилиты (вспомогательные рабочие программы);
- отладчик (средство, помогающее находить и устранять ошибки в программе);
- редактор связей;
- встроенный ассемблер;
- встроенная справочная служба.

Эти инструменты взаимодействуют между собой через обычные файлы с помощью стандартных возможностей файловой системы.

Системы программирования обеспечивают весь спектр задач по обработке информации. С их помощью можно решать вычислительные задачи, обрабатывать тексты, получать графические изображения, осуществлять хранение и поиск данных и т. д. Кроме того, сами системы программирования представляют собой программы, написанные на языках программирования, т. е. созданные с помощью систем программирования. Наиболее популярные системы программирования — Turbo Basic, Quick Basic, Turbo Pascal, Turbo C.

Различают системы общего назначения и языково-ориентированные системы:

- системы общего назначения содержат набор программных инструментов (например, текстовый редактор, редактор связей и т. п.), позволяющих выполнять разработку программ на разных языках программирования. Для программирования на конкретном языке программирования требуются дополнительные инструменты, ориентированные на этот язык;
- языково-ориентированные системы предназначены для поддержки разработки программ на каком-либо одном язы-

ке программирования, причем построение такой среды базируется на знаниях об этом языке.

Приведем основные признаки классификации систем программирования.

По *набору исходных языков* системы программирования принято делить на *одноязыковые* и *многоязыковые*. Отличительной особенностью многоязыковых систем программирования является то, что отдельные части (секции, модули или сегменты) программы могут быть подготовлены на различных языках и объединены во время или перед выполнением в единый модуль.

По *возможности расширения* СП подразделяются на *замкнутые* и *открытые*. В открытую систему можно ввести новый входной язык с транслятором, не требуя изменений в системе.

По реализации процесса *трансляции* СП делятся на *компилирующие* и *интерпретирующие*. С точки зрения выполнения работы компилятор и интерпретатор существенно различаются. В интерпретирующей системе осуществляется покомандная расшифровка и выполнение инструкций входного языка (в среде данной системы программирования); в компилирующей — подготовка результирующего модуля, который может выполняться на ЭВМ практически независимо от среды. После того как текст программы откомпилирован, исходный текстовый файл уже не участвует в дальнейшем построении и запуске программы, в то время как текст программы, обрабатываемой интерпретатором, должен заново переводиться при каждом очередном запуске. Таким образом, интерпретируемые программы проще исправлять и изменять, но откомпилированные программы работают быстрее.

Каждый конкретный язык программирования ориентирован либо на компиляцию, либо на интерпретацию — в зависимости от того, для каких целей он создавался. Например, Pascal обычно используется для решения довольно сложных задач, в которых важна скорость работы программ, поэтому обычно реализуется с помощью компилятора. С другой стороны, Basic изначально создавался как язык для обучения алгоритмизации и программированию. Такая цель предполагала интерпретацию — построчное выполнение программы, чтобы сделать процесс обучения более наглядным. Иногда для одного языка имеется и компилятор, и интерпретатор. В этом случае для разработки и тестирования программы можно воспользоваться интерпретатором, а затем откомпилировать отлаженную программу, чтобы повысить скорость ее выполнения.

Рассмотрим структуру абстрактной многоязыковой, открытой, компилирующей системы программирования и процесс разработки приложений в такой среде (рис. 3.6).

Ввод. Программа на исходном языке (исходный модуль) готовится с помощью текстовых редакторов и в виде текстового файла или раздела библиотеки поступает на вход транслятора.

Трансляция. Трансляция исходной программы есть процедура преобразования исходного модуля в промежуточную, так называемую объектную форму. Трансляция в общем случае включает в себя препроцессинг (предобработку) и компиляцию.

Препроцессинг — необязательная фаза, состоящая в анализе исходного текста, извлечения из него директив препроцессора и их выполнения.

Директивы препроцессора представляют собой помеченные спецсимволами (обычно %, #, &) строки, содержащие аббревиатуры, символические обозначения конструкций, включаемых в состав исходной программы перед ее обработкой компилятором.

Данные для расширения исходного текста могут быть стандартными, определяться пользователем либо содержаться в системных библиотеках ОС.

В качестве примера рассмотрим директивы, определяющие функционирование компилятора в системах программирования Pascal.

Каждая директива компилятора заключается в фигурные скобки и начинается с символа «$», после которого без пробела должно быть указано имя директивы: `{$I+}`, `{$IFDEF}`, `{$ELSE}`.

Различают три вида директив препроцессора:

- ключевые директивы;
- директивы параметров;
- директивы условной компиляции.

Ключевая директива включает или отключает определенные директивой возможности компилятора. Ключевые директивы могут носить глобальный или локальный характер. Глобальные директивы определяют весь процесс компиляции и должны размещаться в тексте программы до начала всех объявлений. Локальные директивы определяют компиляцию только части кода, могут многократно включать или отключать заданные режимы компиляции и располагаться в любых местах текста программы. Примеры ключевых директив:

`{$I+}` или `{$I-}` — соответственно включает или отключает контроль ошибок файлового ввода-вывода;

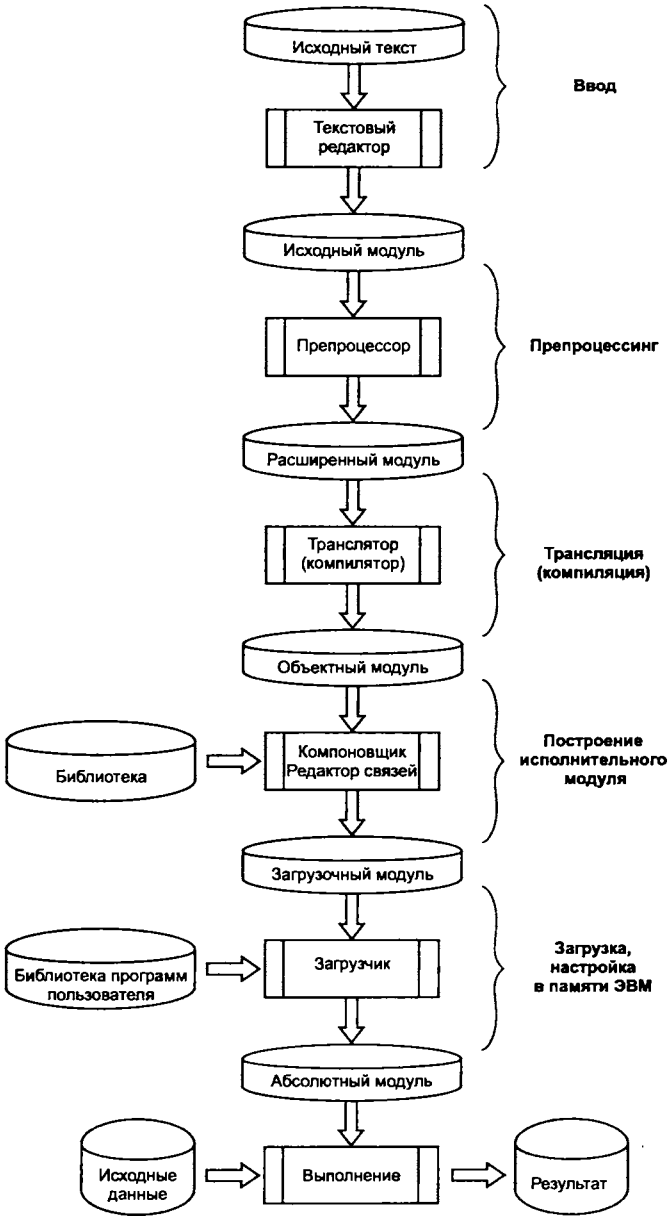


Рис. 3.6. Схема разработки прикладных программ в среде системы программирования

{$R+} или {$R-} — директивы компилятора, включающие и отключающие проверку диапазона целочисленных значений и индексов.

Директивы параметров задают значения различных параметров, например имена файлов, размеры отводимой памяти.

Ключевые директивы и директивы параметров обычно имеют установки по умолчанию.

Директивы условной компиляции позволяют в зависимости от задания тех или иных условий компилировать или исключать из программы на стадии компиляции отдельные фрагменты кода. Условная компиляция дает возможность программисту управлять компиляцией программного кода, например, использование во фрагменте кода директив:

```
{$IFOPT Q+}
...
{$ENDIF}
```

позволит компилировать заключенный между директивами текст программы только в том случае, если включена опция Q проверки переполнения при целочисленных операциях.

Компиляция — в общем случае многоступенчатый процесс, включающий следующие фазы:

- синтаксический анализ — проверка правильности конструкций, использованных программистом при подготовке текста;
- семантический анализ — выявление несоответствий типов и структур переменных, функций и процедур;
- генерация объектного кода — завершающая фаза трансляции.

Выполнение трансляции (компиляции) может осуществляться в различных режимах, установка которых производится с помощью ключей, параметров или опций. Может быть, например, потребовано только выполнение фазы синтаксического анализа и т. п.

Объектный модуль представляет собой текст программы на машинном языке, включающий машинные инструкции, словари, служебную информацию.

Объектный модуль не работоспособен, поскольку содержит неразрешенные ссылки на вызываемые подпрограммы библиотеки транслятора (в общем случае — системы программирования), реализующие функции ввода-вывода, обработки числовых

и строчных переменных, а также на другие программы пользователей или средства пакетов прикладных программ.

Построение исполнительного модуля. Построение загрузочного модуля осуществляют специальные программные средства — редактор связей, построитель задач, компоновщик, основной функцией которых является объединение объектных и загрузочных модулей в единый загрузочный модуль с последующей записью в библиотеку или файл. Полученный модуль в дальнейшем может использоваться для сборки других программ и т. д., это создает возможность наращивания программного обеспечения.

Загрузка программы. Загрузочный модуль после сборки помещается либо в качестве раздела в пользовательскую библиотеку программ, либо в качестве последовательного файла на накопитель на магнитном диске (НМД). Выполнение модуля состоит в загрузке его в оперативную память, настройке по месту в памяти и передаче ему управления. Образ загрузочного модуля в памяти называется абсолютным модулем, поскольку все команды ЭВМ здесь приобретают окончательную форму и получают абсолютные адреса в памяти. Формирование абсолютного модуля может осуществляться как программно (путем обработки командных кодов модуля программой-загрузчиком), так и аппаратно (путем применения индексирования и базирования команд загрузочного модуля и приведения указанных в них относительных адресов к абсолютной форме).

Современные системы программирования позволяют удобно переходить от одного этапа к другому. Это осуществляется в рамках так называемой интегрированной среды программирования, которая содержит в себе текстовый редактор, компилятор, компоновщик, встроенный отладчик и, в зависимости от системы или ее версии, предоставляет программисту дополнительные удобства для написания и отладки программ.

Библиотеки подпрограмм

В п. 1.5 гл. 1 определено понятие подпрограммы как средства многократного использования одного и того же фрагмента алгоритма в разных местах основной программы (например, вычисление одной и той же арифметической функции при разных значениях аргумента).

Однако довольно распространенной является и такая ситуация, когда один и тот же алгоритм — вычисление значений элементарных функций, перевод чисел из одной системы в другую, преобразование символьной информации к строчному или прописному представлению и т. д. — используется при решении самых разных задач. Если один из подобных алгоритмов уже был один раз разработан и реализован при решении некоторой задачи, то возникает естественное желание использовать уже готовую подпрограмму как часть любой другой программы. Таким образом, организовав сбор, хранение и удобное использование готовых подпрограмм, можно существенно упростить и ускорить разработку программ различного назначения, записывая лишь обращения к готовым подпрограммам и заново проектируя уже гораздо меньшую часть кода.

В процессе использования готовых подпрограмм возникает ряд проблем, связанных, с одной стороны, с хранением имеющихся подпрограмм и размещением используемых подпрограмм в памяти ЭВМ, и с другой стороны — с организацией их взаимодействия с основной программой. В связи с этим для обеспечения удобства практической работы выбирается определенная система использования подпрограмм, в которой тем или иным образом эти проблемы решены. Такая система всегда предъявляет определенные требования к подпрограммам с точки зрения их организации и оформления. Подпрограммы, которые удовлетворяют всем требованиям выбранной системы, называются *стандартными*, а организованный соответствующим образом набор таких подпрограмм называется *библиотекой подпрограмм*.

С точки зрения компоновки и последующего взаимодействия с основным программным кодом библиотеки подпрограмм делятся на библиотеки статического вызова (*статические библиотеки*) и библиотеки динамического вызова (*динамические библиотеки*).

В любом случае, когда подпрограмма оказывается недоступной напрямую в исходном файле, компилятор добавляет эту подпрограмму в специальную внутреннюю таблицу, содержащую все внешние идентификаторы. Очевидно, что при этом компилятор должен иметь в своем распоряжении объявление подпрограммы и информацию о ее параметрах.

После компиляции подпрограммы статической библиотеки компоновщик добавляет ее откомпилированный код к исполняемой программе. Получившийся в результате исполнительный

модуль содержит код программы и всех используемых подпрограмм.

В случае динамической компоновки компоновщик просто использует информацию о подпрограмме для настройки соответствующих таблиц в исполняемом файле. Когда исполняемый модуль загружается в память, операционная система загружает также все необходимые динамические библиотеки и заполняет внутренние таблицы программы адресами библиотечных подпрограмм в памяти, после этого программа запускается на исполнение.

Динамическая библиотека (DLL — Dynamic Link Library) представляет собой файл с откомпилированным исполняемым кодом и, таким образом, может использоваться программами, написанными на разных языках программирования.

Отличие динамической библиотеки от исполняемого файла состоит в том, что она не может быть запущена самостоятельно. Динамическая библиотека начинает работать только в случае ее вызова уже работающей программой или библиотекой.

Схема вызова подпрограмм статической и динамической библиотек изображена на рис. 3.7.

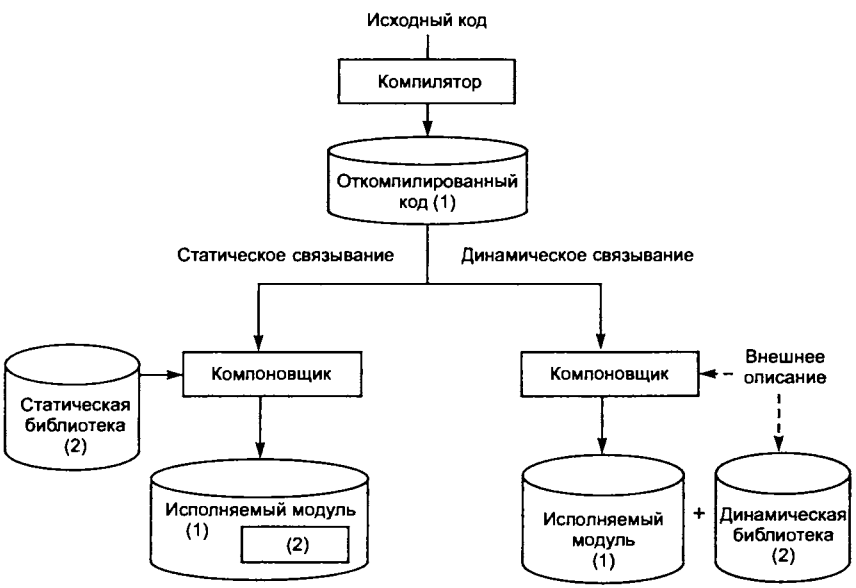


Рис. 3.7. Компиляция статической и динамической библиотек

К преимуществам использования динамических библиотек можно отнести следующие:

- при использовании ее несколькими программами загружается в память только один раз (в отличие от статической компоновки, при которой каждая отдельная программа содержит внутри себя коды всех используемых подпрограмм); при этом сокращается размер исполняемого файла и экономятся системные ресурсы;
- при модификации можно просто заменить старую версию библиотеки новой, не перестраивая при этом готовые программы (если, конечно, изменения не коснулись параметров вызова используемых программами подпрограмм);
- управление возлагается на операционную систему;
- одну и ту же библиотеку могут использовать программы, написанные на разных языках.

Эти общие преимущества оказываются полезными, например, в следующих случаях:

- если несколько программ используют один и тот же сложный алгоритм: сохранение его в динамической библиотеке уменьшит размер исполнительного модуля и сэкономит память, когда несколько программ будут запущены одновременно;
- если необходимо постоянно вносить изменения в программу большого объема: разделение программы на исполняемую часть и динамическую библиотеку позволит модифицировать и затем распространять только измененную часть программы;
- если над одним проектом одновременно работают несколько разработчиков, каждый из которых создает и отлаживает свою часть программы.

Интегрированные среды разработки приложений

Интегрированные среды разработки приложений (Integrated Development Environment — IDE), помимо всех функций системы программирования, содержат инструменты для упрощения конструирования графического интерфейса пользователя и большой спектр сервисов, включающих управление версиями проектов, просмотра и управления информацией, библиотеки классов (для поддержки объектно-ориентированной разработки), мастера создания шаблонов приложений, репозитории проекта и т. п.

Наиболее известные среды разработки приложений — Visual Basic, VisualJ, CBuilder, Delphi, JBuilder.

Система управления версиями позволяет разработчикам следить за изменениями кода программного продукта в ходе его разработки, а также управлять различными его состояниями. Система управления версиями позволяет хранить несколько версий одного и того же продукта, возвращаться (при необходимости) к более ранним версиям, определять, кто и когда внес то или иное изменение.

Многие системы управления версиями предоставляют такие возможности, как:

- создание разных вариантов одной версии разработки (так называемой ветки) с общей историей изменений до точки ветвления и с разными — после нее;
- ведение журнала изменений, в который разработчики могут записывать информацию о том, что и почему они изменили в данной версии;
- контроль прав доступа пользователей на чтение или изменение информации.

Репозитории проекта — место, где хранится и поддерживается информация, связанная с проектом разработки ПС в течение всего его жизненного цикла. Чаще всего данные в репозитории хранятся в виде файлов, доступных для дальнейшего распространения. В современных инструментальных системах репозитории приобретают роль фундамента всей среды разработки.

В средствах разработки приложений может быть выявлено наличие пяти базовых уровней:

- проектирования интерфейсов;
- программирования;
- распространения;
- доступа к внешним источникам данных;
- репозитория.

Уровень разработки интерфейсов — это подсистема средства разработки, предназначенная для создания экранов, окон и меню, с которыми может взаимодействовать пользователь. Обычно доступны функции тестирования, позволяющие увидеть, как работает интерфейс сразу после его создания.

Уровень программирования — это подсистема средства разработки, обеспечивающая создание программного кода. Большинство современных средств разработки управляется событиями, т. е. приложение выполняет некоторое действие в соответст-

вии с поведением интерактивного пользователя. Разработчики связывают программный код с управляющими воздействиями интерфейса, используя редакторы кода.

Уровень распространения предоставляет средства трансляции приложения в нечто, выполняемое клиентом. Средство может создать интерпретируемый код и предоставить его интерпретатор. Однако преимущество отдается средам, создающим исполняемый код.

Уровень доступа к внешним источникам данных действует как посредник между источником данных и средством разработки (а также и приложением после завершения его разработки). Этот уровень обеспечивает управление всеми обращениями к источнику данных. В универсальных средах разработки он независим от типа источника данных.

Уровень репозитория представляет собой дополнительный уровень абстракции над уровнем доступа к базам данных. В разных средствах разработки возможности и емкость репозитория существенно различаются. Некоторые репозитории могут хранить и обрабатывать объекты приложений и даже определять параметры безопасности приложения. Во многих случаях репозитории поддерживают некоторые объектно-ориентированные аспекты средства разработки.

3.4. Средства автоматизации разработки программ

Средства автоматизации разработки (CASE-средства) — это инструментарий для системных аналитиков, разработчиков и программистов, позволяющий автоматизировать процесс проектирования и разработки программного обеспечения. Первоначально под CASE-средствами (Computer Aided Software/System Engineering) понимались средства, применяемые на ранних, наиболее трудоемких процессах жизненного цикла — анализе и проектировании. Международный стандарт ISO/IEC 14102:1995 «Информационные технологии. Руководство по оцениванию и выбору инструментальных CASE-средств» определяет CASE-средство более широко — как программное средство, поддерживающее все процессы жизненного цикла программного обеспечения: анализ и формулировку требований, проектирование ПС, генерацию кода, тестирование, документирование, обеспе-

чение качества, конфигурационное управление и управление проектом.

Таким образом, современные CASE-средства охватывают обширную область поддержки многочисленных технологий проектирования ПС — от средств анализа и документирования до полномасштабных средств автоматизации, покрывающих весь жизненный цикл ПС.

Большинство существующих CASE-средств основано на методах структурного или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между модулями, динамики поведения и архитектуры программных средств.

Наиболее трудоемкими этапами разработки ПС являются этапы анализа и проектирования, в процессе которых CASE-средства обеспечивают качество принимаемых технических решений и подготовку проектной документации. При этом большую роль играют методы визуального представления модели программной системы, что предполагает построение структурных или иных диаграмм в реальном масштабе времени.

Под моделью программной системы в общем случае понимается формализованное описание системы на определенном уровне абстракции. Каждая модель определяет конкретный аспект системы, использует набор диаграмм и документов заданного формата, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями или задачами.

Графические (визуальные) модели представляют собой средства для визуализации, описания, проектирования и документирования архитектуры системы. Состав моделей, используемых в конкретном проекте, и степень их детальности в общем случае зависят от следующих факторов:

- сложности проектируемой системы;
- необходимой полноты ее описания;
- знаний и навыков участников проекта;
- времени, отведенного на проектирование.

На рис. 3.8 изображена связь исходного текста программы с характеристиками объекта автоматизации с помощью диаграмм и спецификации модели программной системы. При этом диаграммы опираются на теоретический фундамент в виде теории множеств, теории графов, матриц и т. п. Наличие теоретической

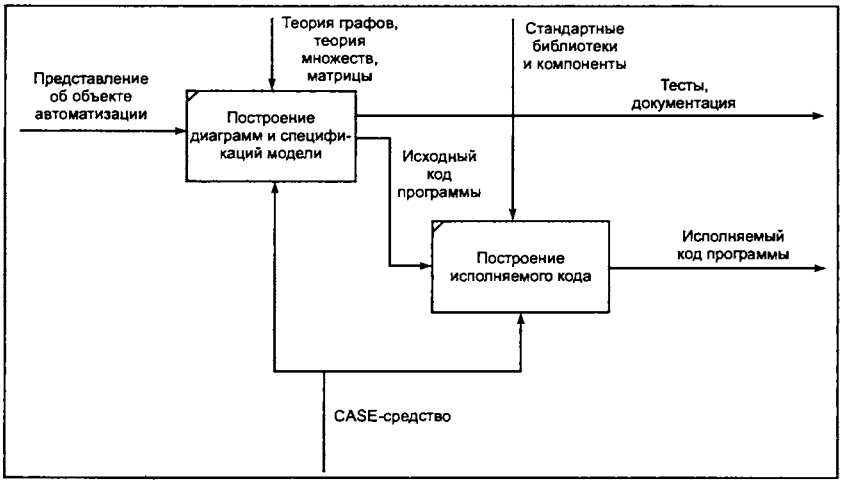


Рис. 3.8. Моделирование программной системы

основы позволяет, во-первых, упростить операции по преобразованию нарисованных на экранах дисплеев диаграмм в память компьютеров и уменьшить объем памяти, необходимой для хранения диаграмм, а во-вторых, реализовать автоматические процедуры преобразования в исполняемый код. Вместе с тем, документирование программ средствами диаграмм и спецификаций создает единый язык общения между программистами, а также между программистами, системными аналитиками и заказчиками.

Классификация CASE-средств

По степени интегрированности CASE-средства могут быть классифицированы следующим образом:

- локальные средства, решающие автономные задачи;
- частично интегрированные средства, охватывающие большинство этапов жизненного цикла ПС;
- полностью интегрированные средства, поддерживающие весь ЖЦ ПС и ысвязанные общим репозиторием.

По функциональности (компонентному составу) выделяют следующие классы:

- средства анализа, предназначенные для построения и анализа моделей предметной области;

- средства анализа и проектирования, поддерживающие наиболее распространенные методологии проектирования и использующиеся для создания проектных спецификаций;
- средства проектирования баз данных, обеспечивающие моделирование данных и генерацию схем баз данных (как правило, на языке SQL) для наиболее распространенных СУБД;
- средства реинжиниринга, обеспечивающие анализ программных кодов и схем баз данных и формирование на их основе различных моделей и проектных спецификаций.

Приведем примеры CASE-средств, наиболее известных в настоящее время: BPwin (инструмент для моделирования бизнес-процессов, поддерживает сразу три нотации моделирования: IDEF0, IDEF3 и DFD); ERwin (средство моделирования баз данных и хранилищ данных); Rational Rose (средство моделирования объектно-ориентированных информационных систем, базирующееся на языке моделирования UML); Oracle Designer (входит в Oracle9i Developer Suite как средство проектирования программных систем и баз данных); ARIS Toolset (средство моделирования, анализа и оптимизации бизнес-процессов).

3.5. Методы и языки моделирования программных систем

В профессиональном программировании на сегодняшний день в основном применяются две методологии моделирования программных систем [9]:

- «классическая» структурная (функциональная) — рассматривает систему в терминах функций и передачи информации между ними;
- объектно-ориентированная — рассматривает структуру взаимодействующих в системе объектов и связи между ними, а поведение системы представляет в терминах обмена сообщениями между объектами.

3.5.1. Структурная методология

Структурная методология использует модели, отражающие:

- функциональную декомпозицию системы;
- последовательность выполняемых действий;

- передачу информации между функциональными процессами;
- отношения между данными.

Охарактеризуем далее наиболее распространенные модели этих групп и языки (графические нотации) их описания.

Функциональная модель IDEF0

Модель IDEF0 является частью семейства стандартов IDEF[1] и представляет собой описание системы в целом как множества взаимозависимых действий или функций, причем IDEF0-функции системы исследуются независимо от объектов, которые обеспечивают их выполнение.

Наиболее часто модель IDEF0 используется при исследовании и проектировании систем на концептуальной стадии разработки, для сбора данных и моделирования процессов «как есть» («as is»). При построении модели необходимо определить:

1. Назначение модели — набор вопросов, на которые должна ответить модель.

2. Границы моделирования — ширину охвата предметной области и глубину детализации.

3. Целевую аудиторию, для нужд которой создается модель.

4. Точку зрения, с которой наблюдается система при построении модели. Точка зрения должна учитывать обозначенные назначение модели и границы моделирования и должна быть неизменной для всех элементов модели.

Главной организационной единицей модели (как и всех моделей, рассматриваемых ниже) является диаграмма. Графический язык модели содержит всего два элемента — блоки (функции) и стрелки (связи).

IDEF0-модель представляет собой иерархическое множество вложенных последовательностей функциональных блоков, поэтому в первую очередь должна быть определена функция, описывающая систему в целом — *контекстная функция*. Далее в процессе построения модели любой блок может быть декомпозирован на составляющие его блоки.

---

[1] IDEF (Integrated DEFinition) — взаимная совокупность методик и моделей концептуального проектирования, разработана в США по программе Integrated Computer-Aided Manufacturing.

Блок на функциональной диаграмме изображается именованным прямоугольником (рис. 3.9). В полное описание блока входит четыре типа стрелок, каждая из которых соединяется с определенной стороной функционального блока:

- I (Input) — вход — потребляемая и/или преобразуемая информация;
- C (Control) — управление — ограничения и инструкции, влияющие на ход выполнения процесса;
- (Output) — выход — информация, получаемая в результате работы функции;
- M (Mechanism) — исполняющий механизм, который используется для выполнения процесса, но остается неизменными.

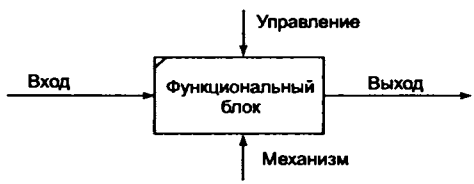


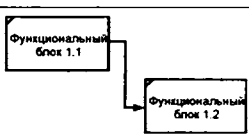
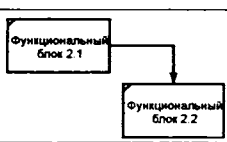
Рис. 3.9. Функциональный блок диаграммы IDEF0

Каждый функциональный блок должен иметь как минимум по одной стрелке входа, выхода и управления.

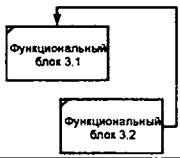
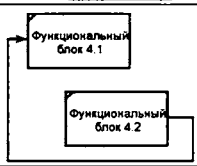
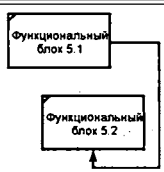
Комбинированные *стрелки* соединяют функциональные блоки и определяют порядок выполнения функций, передачи информации и управления. В табл. 3.4 представлены пять основных видов соединений.

Пример функциональной диаграммы можно видеть на рис. 3.8.

Таблица 3.4. **Виды комбинированных стрелок**

Графика	Название	Назначение
	«Выход — вход»	Одна из функций должна полностью завершиться перед началом другой (выходная информация одной функции служит входом для другой)
	«Выход — управление»	Один блок управляет работой другого

Окончание табл. 3.4

Графика	Название	Назначение
	«Выход — обратная связь на управление»	Зависимый блок формирует обратную связь на управление
	«Выход — обратная связь на вход»	описание циклов повторной обработки
	«Выход — механизм исполнения»	Выход одного блока является инструментом для исполнения другого

Метод моделирования IDEF3

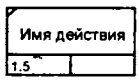
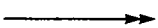
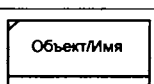
Метод является частью семейства стандартов IDEF и предназначен для построения моделей таких процессов, в которых важно понять последовательность выполнения действий и взаимозависимости между ними. Метод IDEF3 может служить дополнением к методу функционального моделирования IDEF0 (для детализации функциональных блоков IDEF0, не имеющих диаграмм декомпозиции). Основой модели IDEF3 служит так называемый сценарий процесса, который выделяет последовательность действий и подпроцессов анализируемой системы.

Графический язык модели содержит следующие элементы (табл. 3.5):

- действия;
- связи;
- соединения;
- указатели.

Действия изображаются в виде прямоугольника, содержащего имя действия и составной номер действия, который включает номер родительского действия (на рисунке — 1) и непосредственный номер самого действия (на рисунке — 5), разделенные точкой.

Таблица 3.5. **Графический язык модели IDEF3**

Графика	Название	Назначение
	Действие	Моделирует некоторое действие, преобразующее вход в выход. По своему назначению почти идентично функциональным блокам IDEF0 и DFD
	Связь типа «Временное предшествование»	Обозначает, что исходное действие должно завершиться прежде, чем начнется конечное действие
	Связь типа «Объектный поток»	Обозначает, что выход исходного действия является входом конечного действия (очевидно, включает и связь «Временное предшествование»)
	Связь типа «Нечеткое отношение»	Используется, когда отношение между действиями не соответствует ни одному из предыдущих типов. Значение каждой такой связи должно определяться аналитиком отдельно
	Указатель	Специальный элемент, позволяющий комментировать и пояснять другие элементы модели
	И-соединение	Инициирует выполнение конечных действий, т. е. все действия, входящие в И-соединение, должны быть завершены перед выполнением исходящих из него действий
	Эксклюзивное ИЛИ-соединение	Инициирует только одно из исходящих действий (для разворачивающего соединения) после того, как только одно из входящих действий (для сворачивающего соединения) будет выполнено
	ИЛИ-соединение	Предназначено для ситуаций, которые не могут быть описаны двумя предыдущими типами соединений, т. е., может быть, несколько действий должно закончиться (для сворачивающего соединения), прежде чем будет инициировано одно или несколько действий (для разворачивающего соединения)

Связи определяют взаимоотношения между действиями, являются однонаправленными и изображаются стрелками, вид которых соответствует типу связи.

Указатели — специальные элементы, которые ссылаются на другие элементы модели (для привлечения внимания к важным аспектам модели). Указатель изображается в виде прямоугольника, похожего на действие. Имя указателя обязательно включает его тип.

Допустимы следующие типы указателей:

- *объект* (Object) — для описания объекта, принимающего участие в действии;
- *ссылка* (GoTo) — для реализации цикличности выполнения действий;
- *единица действия* (Unit of Behavior — UOB) — для многократного изображения на диаграмме одного и того же действия;
- *заметка* (Note) — для документирования информации общего характера;
- *уточнение* (Elaboration — Elab) — для уточнения или более подробного описания элемента, изображенного на диаграмме.

Соединения используются для описания ветвления процесса. *Разворачивающее соединение* описывает процесс, когда завершение одного действия инициирует начало выполнения сразу нескольких действий. *Сворачивающее соединение* применяется, когда некоторое действие требует предварительного завершения нескольких предшествующих. В табл. 3.5 приведены возможные типы соединений.

На рис. 3.10 приведены примеры сворачивающего ИЛИ-соединения и разворачивающего И-соединения.

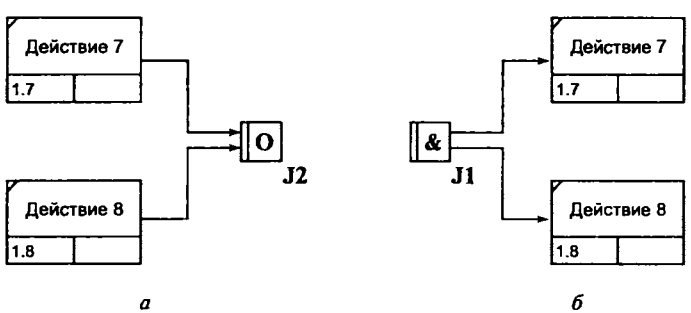


Рис. 3.10. Примеры соединений:
а — сворачивающее; *б* — разворачивающее

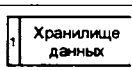
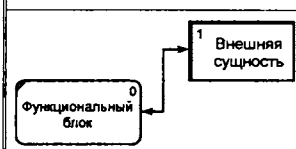
Диаграммы потоков данных (Data Flow Diagrams — DFD)

Диаграммы потоков данных моделируют систему как набор функциональных процессов, связанных потоками данных. Цель такого представления — продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Диаграммы потоков данных содержат (в отличие от функциональной диаграммы IDEF0) два новых типа объектов — хранилища данных и внешние сущности. Эти объекты позволяют определять взаимодействие с частями системы (или другими системами), которые выходят за границы моделирования. Стрелки в DFD отражают такие характеристики системы, как движение объектов (потоки данных), хранение объектов (хранилища данных), источники и потребители данных (внешние сущности).

Для построения DFD традиционно используются две различные нотации, соответствующие методам Йордона — Де Марко и Гейна — Сэрсона. Эти нотации незначительно отличаются друг от друга графическим изображением символов. В табл. 3.6 приведен графический язык нотации Гейна — Сэрсона.

Таблица 3.6. **Графический язык модели DFD (нотация Гейна — Сэрсона)**

Графика	Название	Назначение
	Функциональный блок	Моделирует некоторую функцию, преобразующую вход в выход. По своему назначению почти идентичен функциональным блокам IDEF0 и действиям IDEF3
	Внешняя сущность	Обеспечивают необходимые входы и (или) выходы для функциональных блоков. Одна и та же внешняя сущность может функционировать и как поставщик, и как получатель данных и может повторяться на диаграмме несколько раз
	Хранилище данных	Механизм, который поддерживает хранение данных для их промежуточной обработки
	Потоки данных	Описывают перемещение данных между частями системы. Стрелки, обозначающие потоки, могут быть как однонаправленными, так и двунаправленными, а также могут начинаться и заканчиваться на любой стороне блока

Потоки данных на диаграмме могут представлять ветвление и объединение данных. *Ветвление* обозначает декомпозицию данных, перемещаемых потоком. При этом все полученные после ветвления подпотоки должны быть поименованы (рис. 3.11, *а*). *Объединение* обозначает соединение данных для формирования комплексных объектов данных(рис. 3.11, *б*). Так же, как и при ветвлении, все части комплексного объекта и сам получаемый объект данных должны быть поименованы.

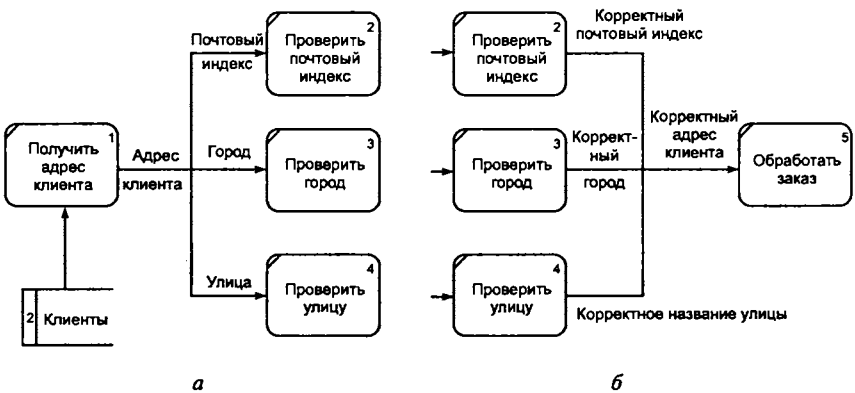


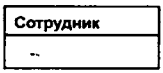
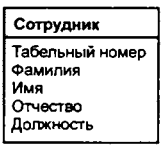
Рис. 3.11. Потоки данных:
а — ветвление; *б* — объединение

Контекстная диаграмма DFD (т. е. диаграмма, описывающая систему в целом) обычно состоит из одного функционального блока и нескольких внешних сущностей, с которыми система обменивается данными. Имя функционального блока на такой диаграмме совпадает с именем системы.

Модель «сущность—связь»

Наиболее распространенным средством моделирования данных (предметной области) является модель «сущность—связь» (Entity-Relationship Model — ERM), впервые предложенная Питером Пин-Шэн Ченом в 1976 г. Эта модель традиционно используется в структурном анализе и проектировании, но, по существу, реализует объектный подход к моделированию предметной области. Одна из разновидностей модели «сущность—связь» используется в нотации IDEF1X, входящей в семейство стандартов IDEF (табл. 3.7).

Таблица 3.7. **Графический язык модели «сущность—связь» (нотация IDEF1X)**

Графика	Название	Назначение
	Сущность	Моделирует класс однотипных объектов, должна иметь наименование, выраженное существительным в единственном числе
	Свойство сущности	Именованная характеристика, являющаяся некоторым атрибутом сущности, выражается существительным в единственном числе (возможно, с характеризующими прилагательными)
	Связь типа «один к одному»	Означает, что один экземпляр первой сущности (левой) связан с одним экземпляром второй сущности (правой)
	Связь типа «один ко многим»	Означает, что один экземпляр первой сущности (левой) связан с несколькими экземплярами второй сущности (правой)
	Связь типа «многие ко многим»	Означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и каждый экземпляр второй сущности может быть связан с несколькими экземплярами первой сущности

Графический язык модели содержит три типа понятий — *сущности, свойства сущностей* и *связи*. Сущность моделирует класс однотипных объектов данных. Так же, как объект, имеет атрибуты, его характеризующие, так же и сущность описывается своими свойствами. Связь сущностей характеризует взаимоотношения между классами объектов.

Сущности, свойства и связи могут быть разных типов. Тип любого элемента диаграммы отображается в его графическом представлении.

Сущности могут быть независимыми и зависимыми. При общем графическом изображении сущности в виде прямоугольника, зависимая сущность очерчена двойной линией.

Свойства сущности (если они приводятся на диаграмме) изображаются в виде овалов или (в зависимости от нотации) перечисляются внутри прямоугольника сущности.

Связи различаются по виду линии, соединяющей сущности (непрерывная линия отображает обязательную связь, пунктир-

ная — необязательную), и по мощности. Мощность связи бывает одного из трех типов: «один к одному», «один ко многим» и «многие ко многим».

На рис. 3.12 представлен фрагмент ER-диаграммы в IDEF1X-нотации логической модели CASE-средства ERwin. На диаграмме представлены сущности (прямоугольники) и связи. Необязательные связи между сущностями *Отдел* и *Сотрудник*, *Сотрудник* и *Подчиненный* имеют мощность «один ко многим» (конец связи «многие» обозначен жирной точкой), а обязательная связь между сущностями *Сотрудник* и *Проект* имеет мощность «многие ко многим».

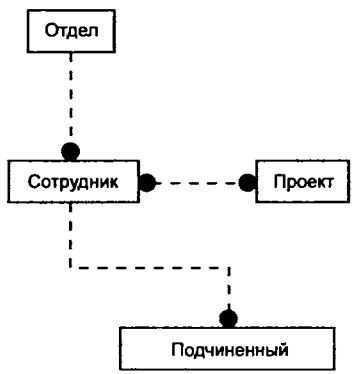


Рис. 3.12. Фрагмент ER-диаграммы

3.5.2. Объектно-ориентированная методология

Концептуальной основой объектно-ориентированного анализа и проектирования программного обеспечения (ООАП) является объектная модель. Ее основные принципы (абстрагирование, инкапсуляция, модульность, наследование) и понятия (класс, интерфейс, объект, атрибут, сообщение) сформулированы Гради Бучем в его фундаментальной книге [7] и последующих работах.

В методах ООАП наглядные модели часто связываются с такими зрительными образами, как «взгляды», направленные на сложную систему с различных точек зрения. Набор из нескольких наглядных моделей создает в сознании специалистов интегральный образ сложной компьютерной системы, которую они совме-

стно проектируют. Вместе с тем наглядные модели служат эффективным средством документирования компьютерных систем и их программного обеспечения, а также языком общения между программистами, системными аналитиками и заказчиками систем.

Язык UML

Большинство современных методов ООАП основаны на использовании языка UML. Унифицированный язык моделирования UML (Unified Modeling Language) представляет собой язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы. Структуру UML составляет стандартный набор диаграмм и нотаций.

Главными в разработке UML были следующие цели:

- предоставить готовый к использованию выразительный язык визуального моделирования, позволяющий разрабатывать осмысленные модели и обмениваться ими;
- предусмотреть механизмы расширяемости базовых концепций языка;
- обеспечить независимость от конкретных языков программирования и процессов разработки;
- интегрировать лучший практический опыт.

В настоящее время UML находится в процессе стандартизации, проводимом организацией по стандартизации в области объектно-ориентированных методов и технологий (Object Management Group — OMG).

Язык UML имеет три основных разновидности понятий: сущности (или предметы), отношения, диаграммы (рис. 3.13).

Сущности (предметы) — это абстракции, которые являются основными элементами UML-модели. В UML определено четыре разновидности сущностей:

- структурные — представляют статические части моделей (понятийные или физические элементы);
- поведенческие — динамические части моделей, представляющие поведения во времени и пространстве;
- группирующие — организационные части моделей («ящики», по которым может быть разложена модель);
- поясняющие — комментирующие части моделей, являются замечаниями, которые можно применить для описания или объяснения любого элемента модели.

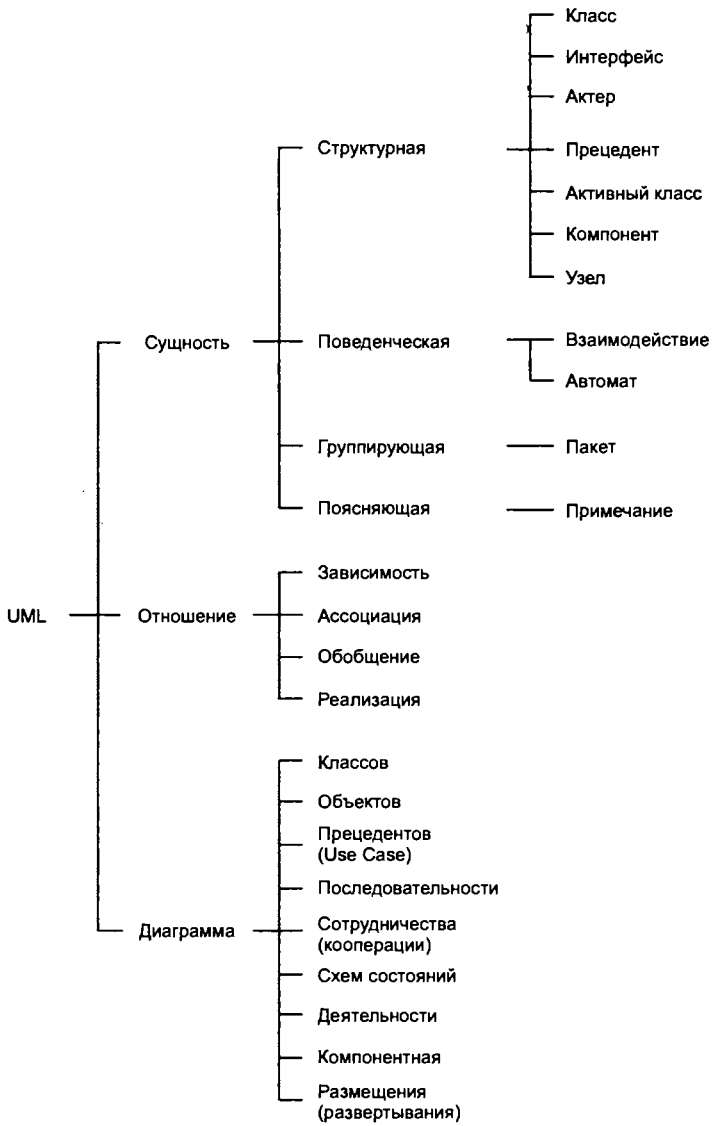
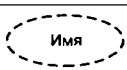
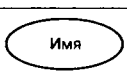


Рис. 3.13. Структура языка UML

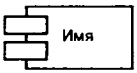
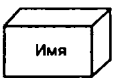
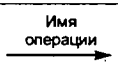
В табл. 3.8 приведены названия, способы графического представления и описания всех сущностей каждой из этих разновидностей.

Отношения — основные связующие строительные блоки при построении UML-модели. Четыре базовых отношения, их графические представления и описания приведены в табл. 3.8.

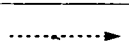
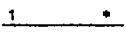
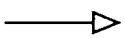
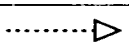
Таблица 3.8. **Сущности и отношения языка UML**

Графика	Название	Назначение
	Структурные сущности	
Имя Свойства Операции	Класс	Описание множества объектов с общими свойствами, операциями, отношениями и семантикой. Класс изображается прямоугольником, разделенным на три поля: – имя класса, однозначно определяющее данный класс среди множества других классов; – общие свойства класса; – типовые операции, выполняемые объектами, принадлежащими данному классу. Класс реализует один или несколько интерфейсов
 IИмя	Интерфейс	Набор операций, определяющий поведение элемента класса (или компонента), видимое извне. Интерфейс задает набор спецификаций операций, а не их реализации. Имя интерфейса обычно начинается на букву I. Интерфейс чаще всего присоединяют к классу или компоненту, который он реализует
 Имя	Кооперация (сотрудничество)	Определяет взаимодействие и объединяет совокупность различных элементов, обеспечивающую коллективное поведение более сложное, чем простая сумма всех элементов. Имеет как структурную, так и поведенческую составляющую. Конкретный класс может участвовать в нескольких кооперациях
	Актер	Набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой. Каждая роль требует от системы определенного поведения, поэтому на UML-диаграммах пиктограммы прецедента и актера обычно располагаются рядом
 Имя	Прецедент (элемент Use Case)	Описание последовательности выполняемых компьютерной системой действий, которые приводят к наблюдаемому актером результату. В модели прецедент применяется для структурирования предметов поведения
 Имя Свойства Операции	Активный класс	Описание множества объектов, которые могут инициировать управляющую деятельность. Активный класс похож на обычный класс, но его объекты действуют одновременно с объектами других классов

Продолжение табл. 3.8

Графика	Название	Назначение
	Компонент	Физическая часть системы, которая соответствует набору интерфейсов и обеспечивает их реализацию. В систему включаются как компоненты — результаты процесса разработки (файлы исходного кода), так и различные разновидности компонентов (например, COM-компоненты)
	Узел	Физический элемент, который представляет собой ресурс, обычно обладающий памятью и возможностями обработки. В узле размещается набор компонентов. Набор компонентов может перемещаться от узла к узлу
Сущности поведения		
	Взаимодействие	Поведение, определяющее набор сообщений, которыми обмениваются объекты (в конкретном контексте) для достижения какой-либо цели. Взаимодействие может характеризовать динамику как группы объектов, так и отдельной операции. Элементами взаимодействия являются сообщения, последовательности действий (поведение) и связи (соединения между объектами)
	Конечный автомат	Поведение, определяющее: – последовательность состояний объекта; – взаимодействия, выполняемые в процессе существования объекта в ответ на события. Элементами конечного автомата являются *состояния*, *переходы* (от одного состояния к другому), *события* (вызывающие переходы) и *действия* (реакции на переходы)
Группирующие сущности		
	Пакет	Общий механизм группировки элементов. В пакет можно поместить структурные и поведенческие предметы и даже другие пакеты. В отличие от компонента, существующего в период выполнения, пакет существует только в период разработки
Поясняющие сущности		
	Примечание	Присоединяется к одному или нескольким элементам диаграммы. Внутри прямоугольника-примечания помещаются комментарии или ограничения, относящиеся к элементу (или нескольким элементам). Комментарий может быть текстовым или графическим

Окончание табл. 3.8

Графика	Название	Назначение
	Отношения	
	Зависимость	Однонаправленное семантическое отношение между двумя сущностями, при котором изменение одной (независимой) сущности вызывает изменение семантики другой (зависимой) сущности. Стрелка графического изображения направлена на независимую сущность
	Ассоциация	Структурное двунаправленное отношение, описывающее совокупность взаимоотношений между объектами. Особая разновидность ассоциации — агрегация — представляет структурное отношение между целым и его частями. Графическое изображение может включать мощности связей и имена ролей.
	Обобщение	Однонаправленное отношение, называемое «потомок-предок», в котором объект «потомок» может быть подставлен вместо объекта родителя («предка»). Потомок наследует структуру и поведение своего родителя. Стрелка всегда указывает на родителя.
	Реализация	Семантическое однонаправленное отношение, которое может устанавливаться: – между интерфейсами и реализующими их классами или компонентами; – между прецедентами (элементами Use Case) и реализующими их кооперациями.

Язык UML обладает механизмами расширения, предназначенными для того, чтобы разработчики могли адаптировать язык моделирования к своим конкретным нуждам, не меняя при этом его метамодель. Наличие механизмов расширения принципиально отличает UML от таких средств моделирования, как IDEF0, IDEF1X, IDEF3, DFD и ERM. Перечисленные языки моделирования не допускают произвольной интерпретации семантики элементов моделей, поэтому, проводя аналогию с языками программирования, их можно определить как сильно типизированные. UML, допуская расширение семантики (в основном за счет стереотипов), таким образом, является слабо типизированным языком.

Стереотип — это новая разновидность уже существующего элемента модели. Стереотипы расширяют нотацию модели и могут применяться к любым элементам модели. Разработчики программных средств могут создавать свои собственные наборы стереотипов, формируя тем самым специализированные под-

множества UML (например, для описания бизнес-процессов, Web-приложений, баз данных и т. д.). Такие подмножества (наборы стереотипов) в стандарте языка UML носят название *профилей* языка.

Диаграммы UML

Диаграмма — графическое представление множества элементов, чаще всего изображаемое в виде связного графа с вершинами-сущностями и ребрами-отношениями. Теоретически диаграммы могут содержать любые комбинации сущностей и отношений, однако на практике применяется, в основном, девять типовых комбинаций.

Диаграмма классов. Диаграмма в общем случае показывает множество классов, интерфейсов, кооперации и отношения между ними и обеспечивает статическое проектное представление системы. Диаграммы классов наиболее часто применяются для моделирования объектно-ориентированных систем. С помощью диаграмм классов составляется словарь системы. Они являются основой для создания диаграмм компонентов и диаграмм развертывания.

Рассмотрим нотации вершин и ребер графа диаграммы классов для простого случая, когда диаграмма показывает классы и отношения между ними.

Основной вершиной диаграммы является *класс* (рис. 3.14, *а*). Для отдельных классов может быть указано только имя, а при дальнейшей разработке добавлены свойства и операции.

Ребра графа диаграммы — ассоциация, агрегация, обобщение и зависимость.

Необязательное имя *ассоциации* может описывать природу отношений. Имени можно придать направление ▶, заданное для чтения имени. Класс, участвующий в ассоциации, играет в ней определенную роль. Роли классов могут быть указаны на концах ассоциации. Мощность ассоциации определяет количество объектов, соединяемых с каждым объектом на другом конце ассоциации, например:

- * — неограниченное количество;
- 1..* — один или более;
- 0..* — ноль или более;
- 1..10 — заданный диапазон;
- 7 — точное количество.

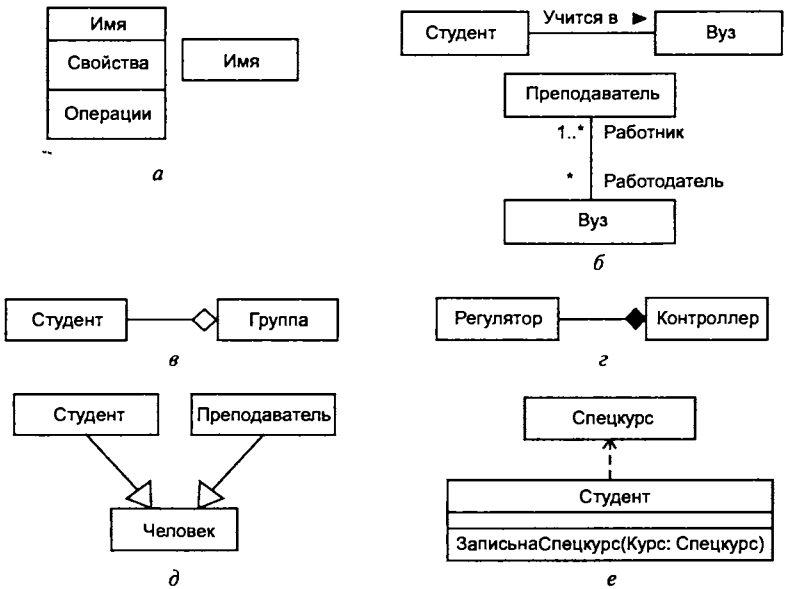


Рис. 3.14. Вершины и ребра диаграммы классов:
а — класс; *б* — ассоциация; *в* — агрегация; *г* — композиция; *д* — обобщение; *е* — зависимость

Агрегация (разновидность ассоциации) определяет отношение «часть—целое». При этом агрегирующая сущность содержит только указатели на части.

Композиция — разновидность ассоциации, определяет непосредственное физическое включение частей в агрегирующую сущность.

Обобщение — отношение между классом и суперклассом. Класс может иметь одного родителя (один суперкласс) или нескольких родителей (несколько суперклассов). Второй случай называют множественным наследованием.

Зависимость — отношение между зависимым и независимым элементом. Обычно операции зависимого клиента вызывают операции независимого либо имеют аргументы (или возвращаемые значения), принадлежащие классу независимого элемента.

Диаграмма схем состояний. Диаграмма моделирует динамику системы — определяет все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий.

Диаграмма схем состояний показывает:

- состояния объектов системы;
- события, вызывающие переход из одного состояния в другое;
- действия, происходящие в результате изменения состояния.

Вершинами графа диаграммы являются состояния (рис. 3.15, *а*). *Состояние* — период в жизни объекта, на протяжении которого он удовлетворяет какому-либо условию, ожидает события или выполняет определенную деятельность. Выделяют *начальное* (рис. 3.15, *б*) и *конечное* (рис. 3.15, *в*) состояния объекта.

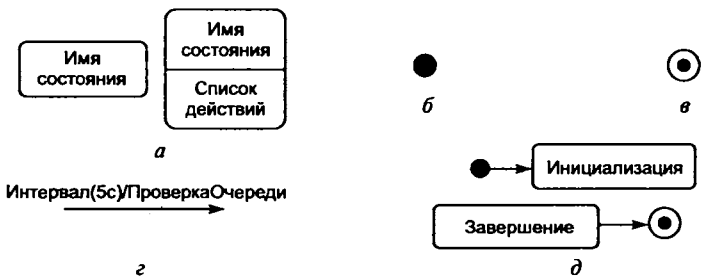


Рис. 3.15. Вершины и ребра графа диаграммы схем состояний: *а* — состояние объекта; *б* — начальное состояние объекта; *в* — конечное состояние объекта; *г* — переход; *д* — инициализация и завершение

Ребра графа диаграммы составляют *переходы* между состояниями (рис. 3.15, *г*). Имя перехода обозначает *событие*, вызывающее изменение состояния («Интервал (5 с)»), и *действие* — набор операций, запускаемых событием («ПроверкаОчереди»). Имя условного перехода заключается в квадратные скобки ([ПереходЕсли]).

Переходы в начальное («Инициализация») и конечное («Завершение») состояния изображаются соответствующими кружками (рис. 3.15, *д*).

Диаграмма деятельности. Диаграмма представляет процесс вычислений и потоки работ. В ней выделяются не обычные состояния объекта, а состояния действий (выполняемых вычислений). При этом предполагают, что процесс вычислений не прерывается внешними событиями.

Диаграммы деятельности особенно полезны в описании поведения, включающего большое количество параллельных процессов.

Вершины графа диаграммы — состояния действия, решения или объединения, линейки синхронизации.

Состояние действия, т. е. выполняемых вычислений (рис. 3.16, *а*), считается атомарным и выполняется за один квант времени (выполнение действия нельзя прервать). Состояние действия не подлежит декомпозиции. Для представления сложного состояния действия (т. е. действия, которое можно разбить на ряд составляющих) используют графическое представление с пиктограммой в правом нижнем углу. Выделяют *начальное* (рис. 3.16, *б*) и *конечное* (рис. 3.16, *в*) состояния действия.

Решение позволяет изобразить ветвление вычислительного процесса. *Объединение* изображает точку слияния альтернативных потоков действий (рис. 3.16, *г*).

Линейка синхронизации (рис. 3.16, *д*) позволяет показать параллельные потоки действий, отмечая точки их синхронизации при запуске действия (момент разделения) и при завершении (момент слияния).

Ребра графа диаграммы — *переходы* между вершинами (рис. 3.16, *е*) — изображаются направленными стрелками. Переходы выполняются только по окончании действия. В вершину «Решение» должна входить одна стрелка, а выходить — несколько. Имена условных альтернатив перехода заключаются в квадратные скобки. В вершину «Объединение» входит несколько стрелок, а выходит — одна.

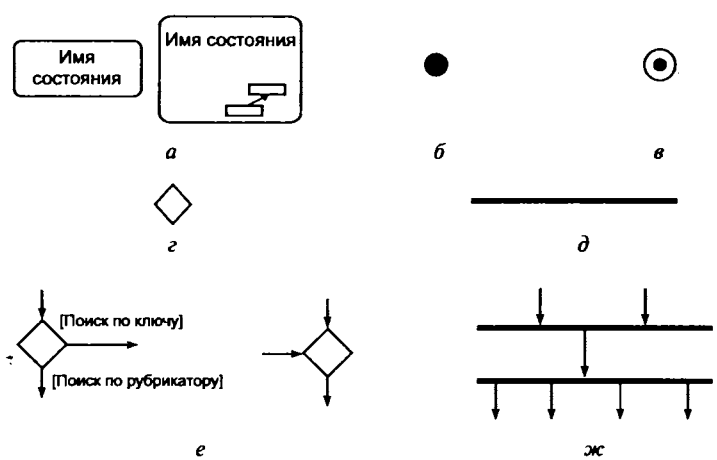


Рис. 3.16. Вершины и ребра графа диаграммы деятельности:
а — состояние действия; *б* — начальное состояние действия; *в* — конечное состояние действия; *г* — решение или объединение; *д* — линейка синхронизации; *е* — переход; *ж* — переходы «Слияние» и «Объединение»

Линейка синхронизации становится вершиной «Слияние» в том случае, если изображает несколько входящих и одну исходящую стрелку, и вершиной «Разделение», если изображает одну входящую и несколько исходящих стрелок (рис. 3.16, *ж*).

Диаграмма сотрудничества (кооперации). Диаграмма представляет собой разновидность диаграммы взаимодействия, предназначена для моделирования динамических аспектов системы и отображает взаимодействие объектов в процессе ее функционирования. Диаграммы сотрудничества концентрируют внимание на связях между объектами.

Вершинами графа диаграммы являются объекты (рис. 3.17, *а*). *Объект* — экземпляр некоторого класса. Имя объекта указывается всегда, свойства — выборочно. Имя объекта должно быть подчеркнуто. Синтаксис представления имени объекта:

```
ИмяОбъекта: ИмяКласса
```

Пропущенное имя объекта означает «любой объект».

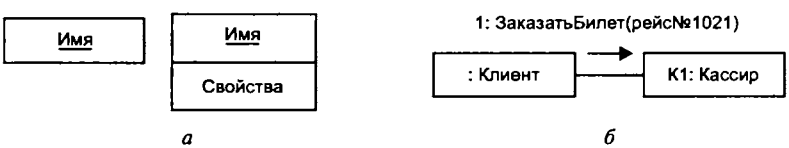


Рис. 3.17. Вершины и ребра графа диаграммы деятельности: *а* — объект; *б* — ассоциация

Синтаксис представления свойства объекта:

```
Имя: Тип = Значение
```

Ребра графа диаграммы — *ассоциации* между классами объектов. Связь между двумя объектами существует только тогда, когда имеется ассоциация между их классами (рис. 3.17, *б*).

Связь между объектами — путь для пересылки сообщения. Сообщение изображается направленной стрелкой, снабженной идентификатором сообщения, имеющим следующий синтаксис:

```
ВозврВеличина:= ИмяСообщ(СписокАргументов)
```

Сообщения в последовательности управления должны быть пронумерованы.

Диаграмма последовательности. Диаграмма последовательности — вторая разновидность диаграммы взаимодействия, отра-

жает временную последовательность событий. Графически диаграмма последовательности — двумерное представление объектов и событий, где объекты размещены на вершине диаграммы вдоль оси X (по возрастанию подчиненности), а сообщения упорядочены по времени вдоль оси Y (рис. 3.18).

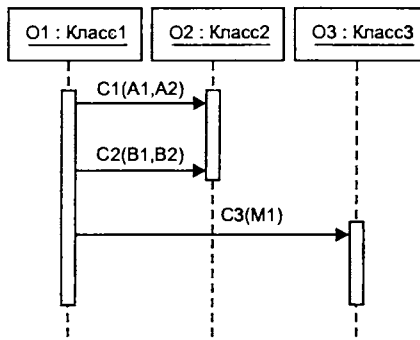


Рис. 3.18. Пример диаграммы последовательности

Графическое изображение объектов и событий такое же, как в диаграмме сотрудничества. Отличительными особенностями диаграммы последовательности являются следующие:

1. *Линия жизни* объекта — вертикальная прерывистая линия, которая обозначает период существования объекта. Уничтожение линии жизни изображается пометкой X в конце линии.

2. *Фокус управления* — высокий тонкий прямоугольник, отображающий период времени, в течение которого объект выполняет действие. Вершина прямоугольника отмечает начало действия, а основание — завершение действия.

Диаграмма прецедентов. Диаграмма прецедентов (*Use Case — диаграмма вариантов использования*) определяет поведение системы и отражает функциональные требования к системе с точки зрения пользователя. Цель построения диаграмм прецедентов — документирование функциональных требований к системе в самом общем виде. Диаграмма должна быть удобна для общения пользователей с разработчиками.

Диаграмма прецедентов определяет системный интерфейс, пользователей и границы системы. В каждой системе обычно есть главная диаграмма прецедентов, которая описывает внешнюю границу системы и основные внешние функции (внешнее поведение) системы. Диаграмма прецедентов может использо-

ваться для разработки тестов и является основой для создания пользовательской документации.

Вершинами графа диаграммы являются актеры и прецеденты.

Актер — роль объекта вне системы, взаимодействующего с частью системы — прецедентом. Имя актера характеризует роль пользователя — физического объекта, который использует систему (рис. 3.19, *а*).

Прецедент (*элемент Use Case*) — описание последовательности действий, которые выполняются системой и производят для актера видимый результат. Каждый прецедент задает определенный способ использования системы. Совокупность всех прецедентов определяет полный набор функциональных возможностей системы (рис. 3.19, *б*).

Ребра графа диаграммы — ассоциация, обобщение, включение, расширение.

Ассоциация — единственный вид отношений между актером и прецедентом, отображает их взаимодействие. Ассоциация может быть помечена именем, ролями и мощностью (рис. 3.19, *в*).

Обобщение — отношение, допустимое либо между актерами, либо между прецедентами (рис. 3.19, *г*). При этом отношение

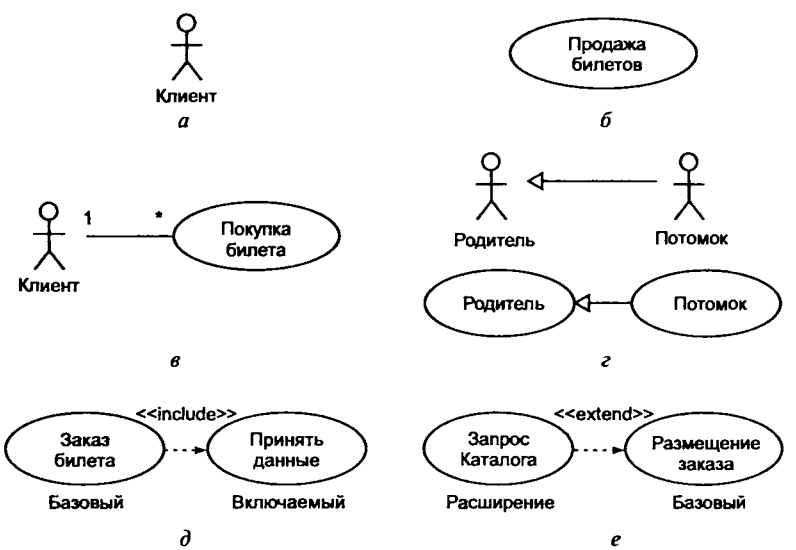


Рис. 3.19. Вершины и ребра графа диаграммы прецедентов:
а — актер; *б* — прецедент; *в* — ассоциация; *г* — обобщение; *д* — включение; *е* — расширение

обобщения между актерами означает, что экземпляр потомка может взаимодействовать с такими же разновидностями прецедентов, что и экземпляр родителя, а отношение обобщения между прецедентами означает, что потомок наследует поведение родителя и даже может дополнить или переопределить поведение родителя. Прецедент, являющийся потомком, может замещать своего родителя в любом месте диаграммы.

Включение — отношение между прецедентами, означающее, что базовый прецедент *явно* включает поведение включаемого прецедента. Включаемый прецедент никогда не используется самостоятельно (рис. 3.19, *д*).

Расширение — отношение между прецедентами, означающее, что базовый прецедент *неявно* включает поведение включаемого прецедента. Базовый прецедент может быть автономен, но при определенных условиях его поведение может расширяться поведением из другого прецедента (но только в заданных точках — точках расширения) — рис. 3.19, *е*.

Диаграмма компонентов. Диаграмма компонентов — первая из двух разновидностей диаграмм реализации, моделирующих физический уровень системы. На диаграммах компонентов изображаются наборы компонентов программной системы и связи между ними.

Диаграммы компонентов используются для моделирования статического представления реализации системы и применяются теми участниками проекта, кто отвечает за компиляцию и сборку системы. Каждый класс модели (или подсистема) преобразуется в компонент исходного кода. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы.

Вершины графа диаграммы — компоненты и интерфейсы.

Компонент — физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов (рис. 3.20, *а*). Таким образом, компонент является физическим фрагментом реализации системы, который заключает в себе программный код (исходный, объектный, исполняемый), сценарные описания или командные файлы операционной системы.

Для обозначения новых разновидностей компонентов в языке используется механизм стереотипов. Некоторые стандартные стереотипы:

1) страницы на языке разметки гипертекста;

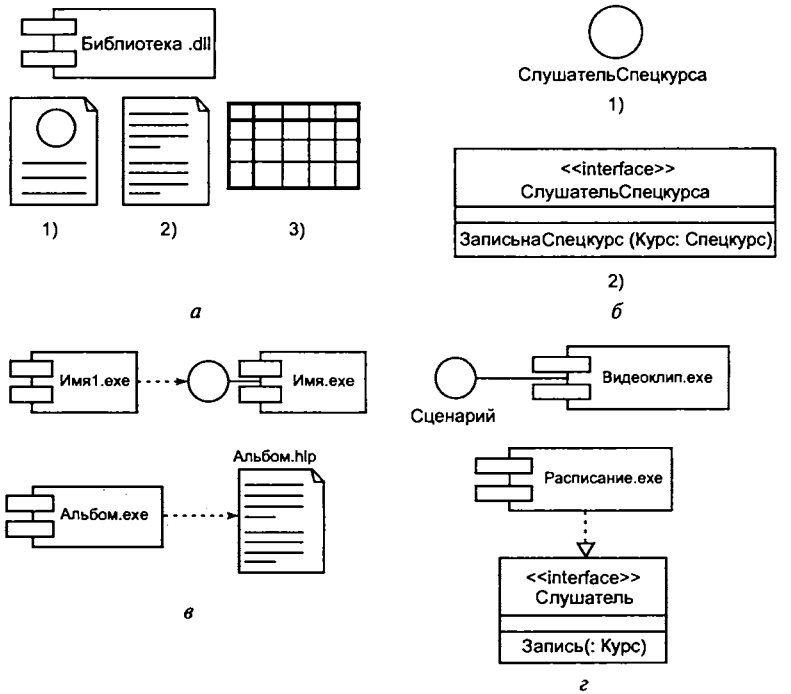


Рис. 3.20. Вершины и ребра графа диаграммы компонентов:
а — компоненты; *б* — интерфейсы; *в* — зависимость; *г* — реализация

2) файлы с исходным кодом или данными;

3) таблица базы данных.

Интерфейс — список операций класса или компонента (рис. 3.20, *б*). Интерфейс подобен абстрактному классу, у которого отсутствуют свойства и подпрограммы выполнения операций, а есть только объявления операций. Таким образом, операции интерфейса только именуют предлагаемые услуги. Все операции интерфейса открыты и видимы клиенту.

Различают два способа представления интерфейса:

1) в виде пиктограммы (свернутый способ);

2) в виде объекта с объявлением операций (развернутый способ).

Ребрами графа диаграммы являются зависимость и реализация.

Зависимость отображает отношение между компонентом и интерфейсом или компонентами. Компонент, использующий интерфейс, связан с ним отношением зависимости (рис. 3.20, *в*).

Реализация отображает отношение между интерфейсом и компонентом, который его реализует. При этом пиктограмма интерфейса соединяется с компонентом, его реализующим, простой линией (рис. 3.20, *г*).

Диаграмма размещения (развертывания). Эта диаграмма — вторая из двух разновидностей диаграмм реализации — отражает физические взаимосвязи между программными и аппаратными компонентами системы.

Диаграммы размещения используются для моделирования статического представления того, как размещается система.

Вершины графа диаграммы — узлы.

Узел — физический элемент, существующий в период работы системы. Он представляет собой вычислительный ресурс, обладающий памятью и, возможно, способностями обработки. Как и класс, узел может иметь дополнительную секцию, отображающую размещаемые в нем элементы. Узел также может содержать компоненты и объекты (рис. 3.21, *а*).

Ребрами графа диаграммы являются ассоциации и зависимости.

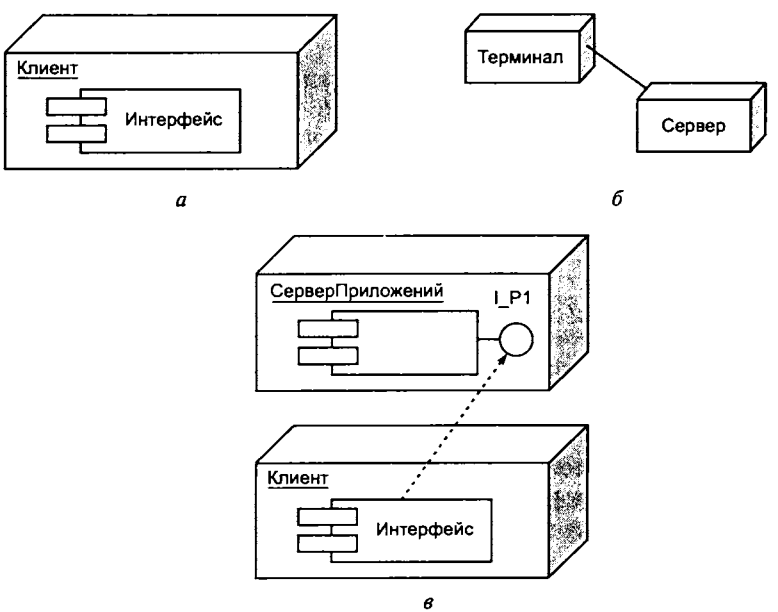


Рис. 3.21. Вершины и ребра графа диаграммы размещения:
а — узел; *б* — ассоциация; *в* — зависимость

Ассоциация в этом случае отображает отношение между узлами диаграммы (рис. 3.21, *б*).

Зависимость на диаграмме размещения отображает отношение между компонентами узлов (прямо или через интерфейс) (рис. 3.21, *в*).

3.6. Основные понятия и принципы тестирования и отладки ПС

Тестирование — процесс выполнения программ с целью обнаружения факта наличия ошибок.

К наиболее распространенным ошибкам общего (несинтаксического) характера, остающимся в программах после выполнения синтаксического контроля, можно отнести следующие:

- логические ошибки (например, неполный учет возможных условий или неверное указание ветви алгоритма после проверки условия);
- ошибки при программировании циклов (например, неверные границы начала и конца);
- ошибки при работе со структурами данных (например, отсутствие исходной инициализации или обнуления элементов структуры);
- ошибки в описании переменной (например, отсутствие инициализации переменной);
- ошибки ввода-вывода.

Организация процесса тестирования

Тестирование начинается с разработки множества тестов и их исполнения на основе одной из выбранных методик. Каждый тест (или тестовый вариант) задает как минимум:

- набор исходных данных и условий для запуска программы;
- набор ожидаемых (эталонных) результатов работы программы.

Набор тестов, разработанных для программы и покрывающих каждый ее линейный участок, называют *тестовым покрытием*. Тестовое покрытие позволяет также определить участки кода, пропущенные при тестировании.

Целью проектирования тестовых вариантов является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

Тестирование обеспечивает:

- обнаружение ошибок в программе;
- демонстрацию выполнения программой заданных функций в соответствии со спецификацией;
- демонстрацию выполнения требований к характеристикам программы;
- отображение надежности как индикатора качества программы.

При проведении тестирования важно помнить, что тестирование никогда не может показать отсутствия ошибок и дефектов — каждый тест показывает только наличие ошибки.

Полную проверку программы гарантирует так называемое *исчерпывающее тестирование.* Количество тестов при исчерпывающем тестировании определяется требованием проверки всех наборов исходных данных и всех возможных вариантов их обработки. Такое тестирование во многих случаях не представляется возможным по причине ресурсных ограничений (прежде всего, ограничений по времени).

Для оценки результатов тестирования используются эталонные файлы, которые содержат ожидаемые результаты работы программы (эталонные результаты). Эталонные результаты могут быть получены, например, вычислением вручную, путем использования справочников или результатов, полученных с помощью другой программы.

Сравнение результатов тестирования с эталонными показывает либо правильность работы программы, либо наличие ошибки. В случае расхождения результатов тестирования начинается диагностика возникшей проблемы. Если с помощью имеющихся тестов локализовать ошибочную ситуацию невозможно, требуется подготовка дополнительных тестов. После локализации и исправления ошибки разработка дополнительных тестов может потребоваться для проверки внесенных изменений. Если после очередного тестирования полученные результаты совпадают с эталонными, проводится оценка достаточности полноты тестирования. Взаимодействие процессов тестирования и отладки схематически представлены на рис. 3.22 [21].

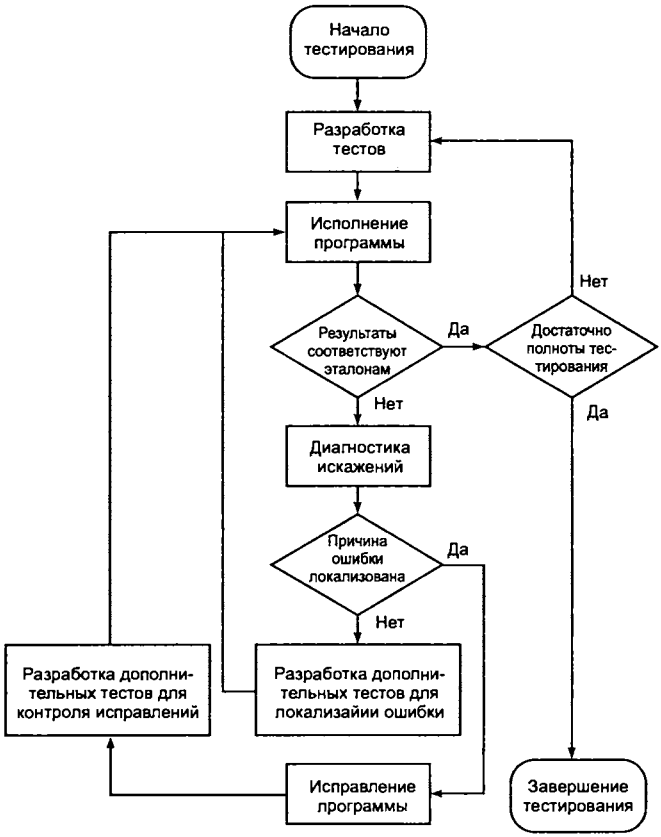


Рис. 3.22. Взаимодействие процессов тестирования и отладки

На рис. 3.23 представлена функциональная диаграмма процесса тестирования, отражающая информационные потоки между функциональными блоками.

На входе процесса тестирования три потока:

- текст программы;
- исходные данные для запуска программы;
- ожидаемые результаты.

После выполнения тестов и оценки ожидаемых результатов формируется прогноз качества и надежности ПС. Если регулярно встречаются серьезные ошибки, приводящие к проектным изменениям, то уровень качества и надежность ПС признаются неудовлетворительными и констатируется необходимость усиле-

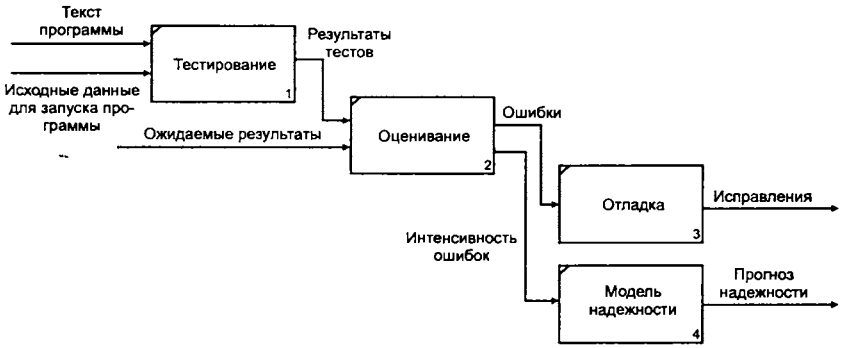


Рис. 3.23. Функциональная диаграмма процесса тестирования

ния тестирования. С другой стороны, если функции ПС реализованы правильно, а обнаруженные ошибки легко исправляются, может быть сделан один из двух выводов: качество и надежность ПС удовлетворительны или разработанные тесты не способны обнаруживать серьезные ошибки.

Тот факт, что тесты, выполняемые в процессе тестирования, не обнаруживают ошибок, на самом деле может говорить о том, что тестовые варианты недостаточно продуманы и что в ПС существуют нераспознанные ошибки. Такие ошибки будут, в конечном итоге, обнаруживаться пользователями и устраняться разработчиками на этапе сопровождения (когда стоимость исправления возрастает в 60—100 раз по сравнению с этапом разработки).

Результаты, накопленные в ходе тестирования, могут оцениваться и более формальным способом. Для этого используют модели надежности ПС, выполняющие прогноз надежности по реальным данным об интенсивности ошибок.

Стратегии тестирования программных продуктов

Существуют две основные стратегии тестирования:

- тестирование программы как *«черного ящика»* (функциональное тестирование), при котором программа рассматривается как объект, внутренняя структура которого неизвестна;
- тестирование программы как *«белого ящика»* (структурное тестирование) подразумевает знание исходного кода программы и полный доступ к нему.

При тестировании программы как *«черного ящика»* исследуется работа каждой функции программы в соответствии со спецификацией. Основное место приложения тестов «черного ящика» — интерфейс ПС.

Тесты функционального тестирования демонстрируют:

- выполнение функций программы;
- корректность ввода исходных данных;
- процесс получения результатов.

При тестировании «черного ящика» принимаются во внимание специфицированные системные характеристики программ, а их внутренняя логическая структура игнорируется. Исчерпывающее тестирование, как правило, невозможно. Например, если в программе 10 входных величин и каждая принимает по 10 значений, то потребуется 10^{10} тестовых вариантов. Тестирование «черного ящика» не реагирует на многие программные ошибки.

Тестирование *«белого ящика»* исследует корректность работы внутренних элементов программы и связей между ними. Объектом тестирования является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. Обычно анализируются управляющие связи элементов, реже — информационные связи. Тестирование по принципу «белого ящика» характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Исчерпывающее тестирование также затруднительно.

Отладка программы

Когда фиксируется ошибка — начинается отладка. *Отладка* — процесс локализации и устранения ошибок. Такой процесс носит творческий характер, плохо формализуем и непредсказуем по времени. Заранее неизвестно, сколько потребуется времени на поиск места дефекта и исправление ошибки. Такая неопределенность приводит к большим трудностям в планировании действий. Тем не менее основной идее отладки (базирующейся на анализе данных) можно придать вид следующего алгоритма:

1. Изучить исходные и полученные результирующие данные.
2. Сформулировать некоторую гипотезу, которая объясняет получение таких результирующих данных.
3. Подготовить новые исходные данные и провести эксперимент, который позволит доказать или опровергнуть гипотезу.

3.7. Принципы разработки графического интерфейса

Концепции графического интерфейса предполагают, что объекты приложений на рабочем столе могут быть визуально представлены в виде окон или пиктограмм, а управление работой приложения и содержимым или формой представления отображаемой информации осуществляется посредством элементов управления.

Обычно интерфейс приложения ориентирован на обработку некоторого центрального объекта, называемого *документом*. Так, например, для текстового редактора документом является текстовый файл, для графического редактора — файл, хранящий изображение, и т. п. В общем случае документ не является однородным (например, текстовый документ формата DOC может содержать различные OLE-объекты — таблицы, диаграммы, формулы) и не обязательно состоит из одного файла.

По количеству одновременно обрабатываемых документов приложения делятся на однодокументные (SDI) и многодокументные (MDI).

Однодокументное приложение в текущий момент времени может работать только с одним центральным объектом — документом. Однако в обработке документа может участвовать несколько окон: помимо главного окна приложения могут открываться вспомогательные, при этом каждое окно однодокументного приложения является самостоятельным и визуально отделено от других. Примером такого приложения служит среда Delphi, описанная в гл. 4 настоящего пособия.

Многодокументное приложение допускает одновременное открытие для обработки нескольких документов. В этом случае одно (главное) окно управляет обработкой различных документов в дочерних окнах, размещаемых в пределах главного окна. Примерами таких приложений служат текстовый процессор Microsoft Word и табличный процессор Excel.

В общем случае окна, предоставляющие доступ к различным типам информации, делятся на первичные и вторичные.

Первичные окна

Первичное окно обеспечивает полное функциональное взаимодействие с центральным объектом приложения. Типовое первичное окно представляет собой ограниченную рамкой область

экрана, снабженную строкой заголовка, в которой идентифицируется информация, отображенная в окне. Первичное окно в строке заголовка содержит уменьшенную копию пиктограммы приложения или объекта, к которому оно относится. В строке заголовка расположены кнопки управления окном, обеспечивающие выполнение операций сворачивания, изменения размера, закрытия. Если размер отображаемой в окне информации превышает его размеры, окно дополняется полосами прокрутки. Для каждого первичного окна на системной Панели задач создается своя кнопка входа. Основные операции с первичными окнами приведены в табл. 3.9.

Таблица 3.9. **Основные операции с окнами**

Операция	Назначение
Открытие/Разворачивание	Окно появляется на экране, автоматически становится активным и располагается на самом верхнем уровне. Операция открытия окна связана с запуском приложения или с активизацией объекта и сопровождается появлением кнопки входа окна на Панели задач. Разворачивание окна делает окно видимым и активным при нажатии на кнопку входа
Закрытие/Сворачивание	Окно приложения или объекта исчезает с экрана. В случае операции закрытия кнопка входа окна удаляется с Панели задач
Изменение состояния (активное — неактивное)	В активном окне пользователь в текущий момент выполняет некоторую последовательность действий. Активное окно обычно расположено «поверх» других окон, а его заголовок выделен специальным цветом. В каждый момент времени активным может быть только одно окно
Изменение размера	Окно может быть развернуто на весь экран, свернуто до кнопки входа на Панели задач или может занимать только часть экрана. Увеличить или уменьшить размеры окна можно также, «растягивая» его границы (зафиксировав курсор мыши на одной из границ)
Перемещение	Окно можно перемещать на области экрана (например, зафиксировав курсор манипулятора «мышь» на строке заголовка окна)
Разбиение	Окно может разделяться на две и более независимые области, называемые подокнами (например, для одновременного просмотра двух частей одного и того же документа или для отображения одной и той же информации в различных формах)

Вторичные окна

Вторичное окно предназначено для отображения или ввода дополнительной информации, управляющей обработкой объектов, отображенных в первичном окне. Стандартное вторичное окно содержит строку заголовка и ограничено рамкой. Открытие вторичного окна не сопровождается появлением кнопки входа на Панели задач. Заголовок вторичного окна не содержит пиктограммы. Для вторичных окон не используются операции разворачивания и сворачивания: размеры вторичных окон обычно не изменяются пользователем.

Вторичное окно может быть независимым или модальным.

Независимое вторичное окно позволяет пользователю переключаться на другие (первичные или вторичные) окна и взаимодействовать с ними.

Модальное вторичное окно не позволяет пользователю переключаться на другие окна до тех пор, пока не закончена работа в модальном окне и оно не будет закрыто.

Среди вторичных окон выделяют следующие типы:

Панель свойств (Property Sheet) — независимое вторичное окно, отображающее доступные пользователю свойства объекта (причем пользователю может быть не дана возможность изменять какие-либо из них). Обычно содержит кнопки **OK**, **Отмена**, **Применить** (рис. 3.24).

Панель контроля параметров (Property Inspector) — модальное вторичное окно, связанное с тем объектом, свойства которого отображает. Внесенные пользователем изменения параметров сразу же применяются к объекту (рис. 3.25).

Диалоговая панель (Dialog Box) — модальное вторичное окно, обеспечивающее диалог между пользователем и приложением. Содержит кнопки **OK**, **Отмена** (или их заменяющие) (рис. 3.26).

Палитра (Palette) — независимое вторичное окно, которое содержит набор взаимосвязанных элементов управления (например, панель инструментов — рис. 3.27).

Сообщение (Message Box) — вторичное окно, предназначенное для вывода сообщений пользователю. Может быть как независимым, так и модальным. Обычно окно сообщения содержит графический символ, обозначающий его тип, и текст сообщения (рис. 3.28).

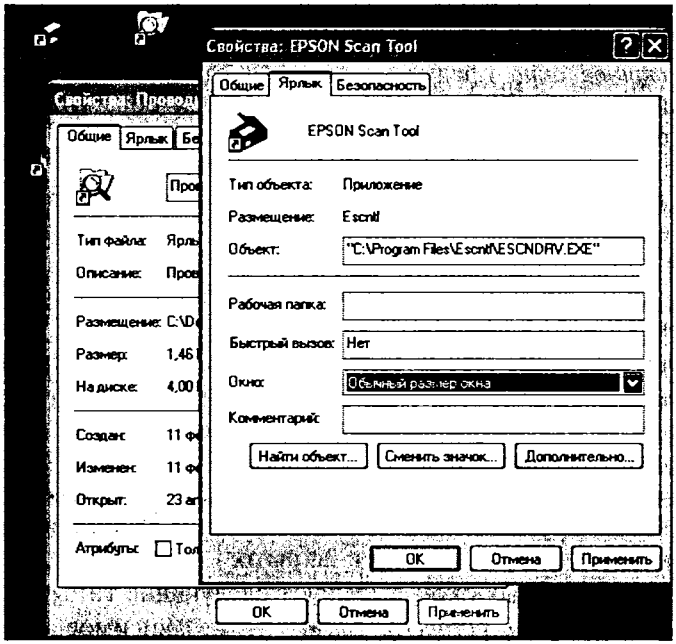


Рис. 3.24. Панель свойств

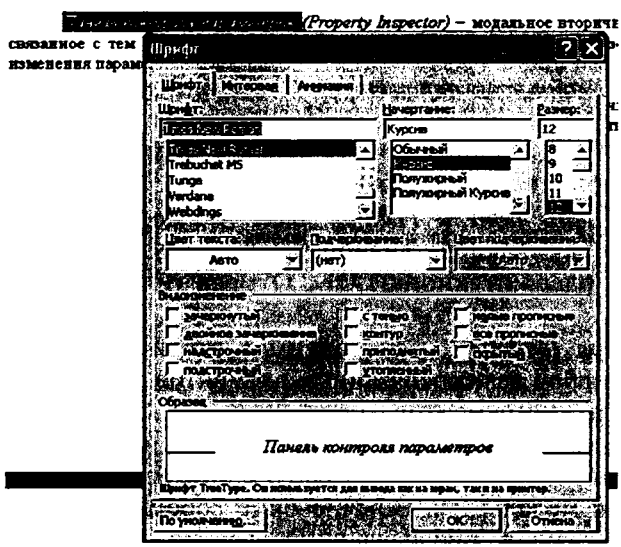


Рис. 3.25. Панель контроля параметров

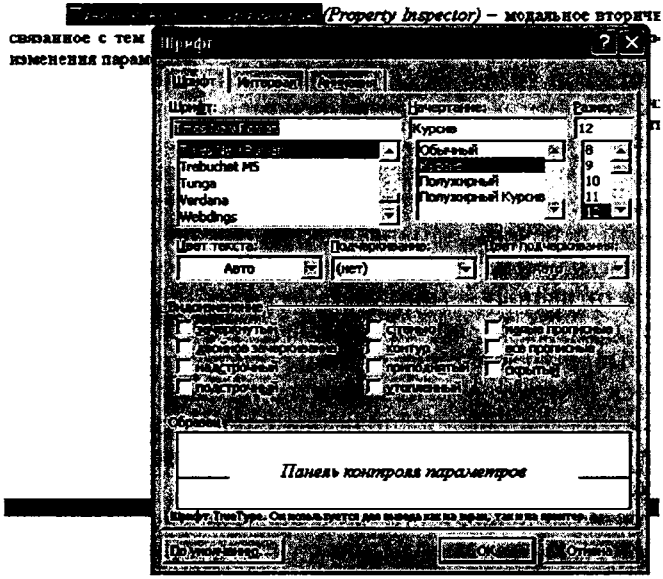


Рис. 3.26. Диалоговая панель

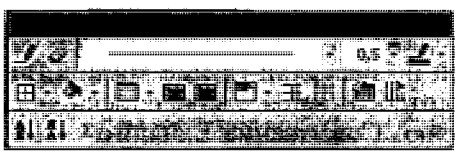


Рис. 3.27. Палитра

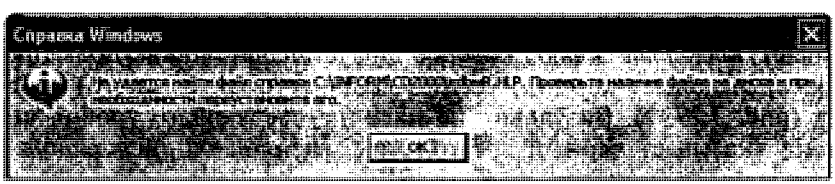


Рис. 3.28. Окно сообщения

Элементы управления

Элементы управления — компоненты графического интерфейса, которые предоставляют пользователю возможность изменять содержимое или форму представления отображаемой информации, а также управлять работой приложения.

Каждый стандартный элемент управления обеспечивает определенный способ взаимодействия пользователя с приложением и имеет свой графический образ, поэтому при создании собственных элементов управления рекомендуется учитывать существующие соглашения.

Рассмотрим некоторые группы стандартных элементов.

Меню. Элемент управления меню всегда содержит структурированный перечень команд, доступных пользователю при работе с приложением. Набор команд меню может меняться в зависимости от выполняемого шага задания или от объекта, с которым в настоящее время работает пользователь. Команды, недоступные пользователю в конкретной ситуации, визуально выделяются (обычно обесцвечивается название команды).

Меню предоставляет пользователю возможность выбора средств для решения задачи и не требует при этом знания синтаксиса команд.

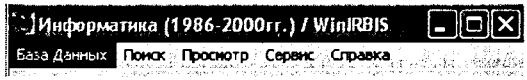
Главное меню первичного окна представляет собой линейную последовательность команд (или групп команд) и располагается в верхней части окна непосредственно под полосой заголовка (рис. 3.29, *а*).

Выбор отдельной команды линейной последовательности инициирует отображение *выпадающего меню* и обеспечивает доступ к следующей по иерархии группе команд (рис. 3.29, *б*). Выпадающее меню отображается в виде панели — столбца с перечнем пунктов меню. Содержание главного меню и связанных с ним выпадающих меню определяется функциональным назначением приложения.

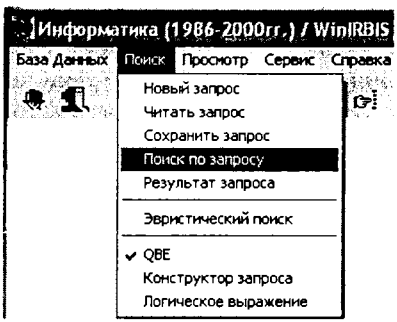
Следующий иерархический уровень выбора команд обеспечивает *каскадное меню*, которое представляет собой подменю, «раскрывающее» команду более высокого (родительского) уровня (рис. 3.29, *г*). Визуально на наличие следующего уровня команд указывает черный треугольник, размещающийся рядом с родительским пунктом меню.

Использование каскадного меню, с одной стороны, предоставляет пользователю возможность дополнительного выбора, не увеличивая размеров родительского меню, но, с другой стороны, ведет к усложнению интерфейса.

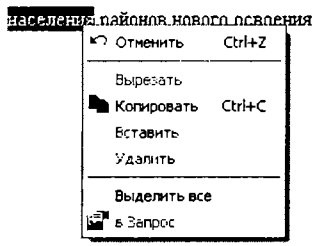
Сложившиеся правила работы с графическим интерфейсом требуют использования наряду с главным меню *всплывающего меню* (рис. 3.29, *в*). Такое меню ориентировано на работу с конкретным объектом и отображается в текущей позиции (по



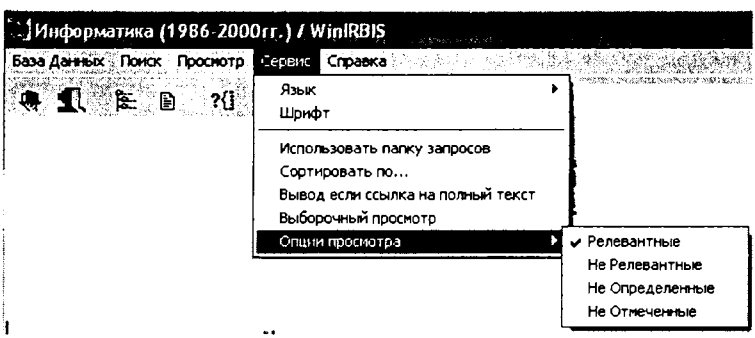
а



б



в



г

Рис. 3.29. Меню:
а — главное меню первичного окна; *б* — выпадающее меню; *в* — всплывающее меню; *г* — каскадное меню

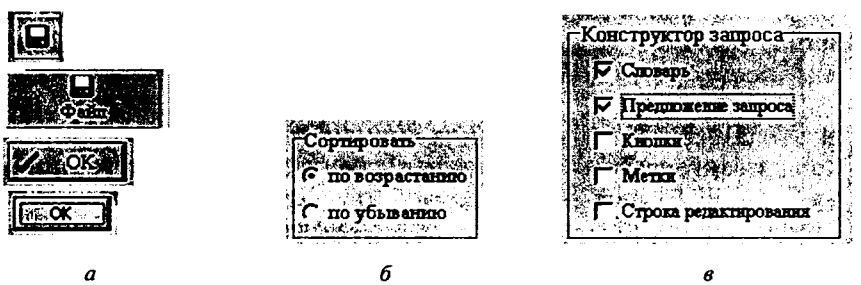
положению курсора), это избавляет пользователя от необходимости перемещать курсор в область меню или панели инструментов. Всплывающее меню содержит только команды, допустимые для указанного объекта в текущей ситуации, сокращая тем самым число команд, среди которых пользователь должен сделать выбор.

Кнопки. Кнопка — элемент графического интерфейса, служащий для запуска на исполнение какого-либо действия или

для изменения свойств объектов. Различают три основных вида кнопок:

- кнопки управления (Command Buttons);
- кнопки установки параметров (Option Buttons, Radio Buttons);
- кнопки независимого выбора или флажки (Check Boxes).

Кнопка управления (или просто кнопка) предназначена для запуска связанной с ней команды или операции. Такая кнопка обычно имеет прямоугольную форму и содержит в качестве метки поясняющий текст, графическое изображение или одновременно и то и другое (рис. 3.30, *а*). Результат действия, запускаемого при «нажатии» кнопки, проявляется немедленно и непосредственно влияет на текущую ситуацию. Если в текущий момент выполнения задания действие, обозначаемое кнопкой, недоступно, метка кнопки обесцвечивается.



а *б* *в*

Рис. 3.30. Кнопки:
а — управления; *б* — установки параметров; *в* — независимого выбора

Кнопки установки параметров (или переключатели) предназначены для выбора единственного варианта из предлагаемого множества взаимоисключающих альтернатив (рис. 3.30, *б*). Так как в любой группе переключателей может быть выбран только один, при проектировании таких переключателей требуется явная группировка по типам объектов или свойств объектов. Как правило, переключатели используются для выбора одного из указанных значений какого-либо свойства объекта и таких значений должно быть не менее двух, так как если кнопка выбрана, то отменить ее можно только выбором другой кнопки.

Кнопки независимого выбора (или флажки) используются для отображения независимых вариантов выбора (рис. 3.30, *в*). Так

же, как и переключатель, флажок может находиться в одном из двух состояний: «установлен» или «снят», но в отличие от переключателей в группе флажков может быть одновременно установлено несколько (или, например, все сняты).

Списки. Списки предназначены для управления выбором требуемых объектов или свойств объектов. Использование списка целесообразно в тех случаях, когда количество возможных вариантов выбора велико, либо когда перечень вариантов может изменяться.

Элементы списка представляются как в текстовой, так и в графической форме. Списки могут различаться способом отображения содержимого и типом выбора, который они поддерживают. Как элемент интерфейса, список обычно используется не только для визуального отображения сделанного пользователем выбора, но и для поддержки связанных с выбором действий. Графически выбор в списке изображается альтернативным цветовым выделением.

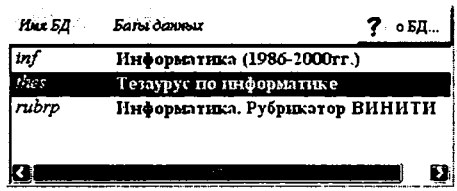
Список единичного выбора (Single Selection List Box) используется для выбора только одной строки в списке и тем самым аналогичен группе переключателей (рис. 3.31, *а*). Однако список в этом случае позволяет более эффективно оперировать большим количеством пунктов, возможно, меняющих свое название, используя ограниченную область окна.

Выпадающий список (Drop-down List Box) аналогичен по возможностям списку единичного выбора, но при этом отображается на экране только по требованию пользователя. На рис. 3.31, *в* изображен выпадающий список в свернутом и развернутом виде.

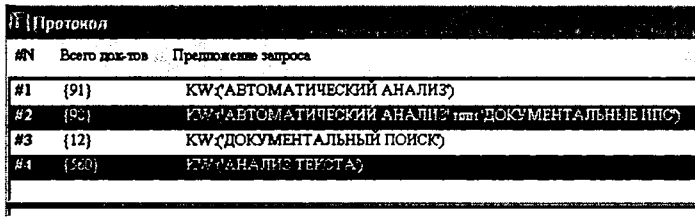
Расширенный список (Extended List Box) и *список множественного выбора* (Multiple Selection List Box) обеспечивают пользователю выбор более чем одной строки (пункта) (рис. 3.31, *б*). При этом поддерживается стандартная техника непрерывного и непересекающегося выбора (т. е. может быть выбран как отдельный пункт, так и непрерывная область).

Модифицируемый список (List View Control) представляет собой форму расширенного списка, отображающую набор пунктов, каждый из которых представлен пиктограммой и текстовой меткой (рис. 3.31, *г*). Содержимое модифицированного списка может быть представлено в разных форматах:

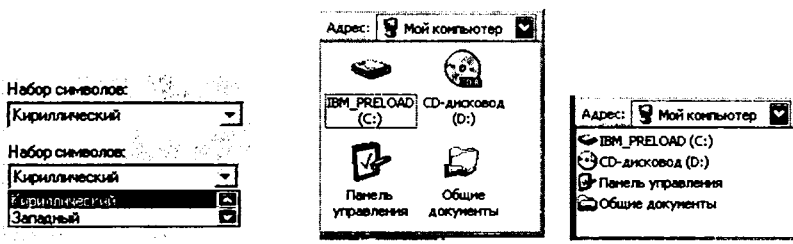
- полноразмерными пиктограммами с расположенным под ними текстом;



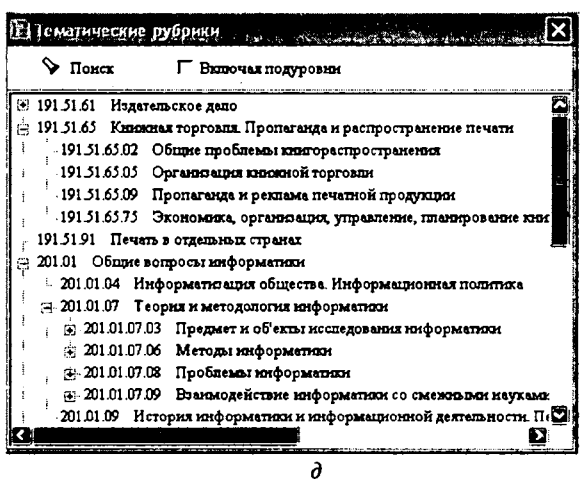
а



б



в *г*



д

Рис. 3.31. Списки:
а — список единичного выбора; *б* — список множественного выбора; *в* — выпадающий список; *г* — модифицируемый список; *д* — модифицируемое дерево

- пиктограммами маленького формата с текстом, расположенным справа;
- пиктограммами маленького формата с текстом, расположенным справа, упорядоченными в виде столбца заданного формата (списком);
- таблицей, самый левый столбец которой содержит пиктограммы и текст, а остальные — информацию, формируемую приложением.

Модифицируемое дерево (Tree View Control) является частным случаем модифицируемого списка, когда содержимое отображается с учетом иерархических отношений между пунктами (рис. 3.31, *д*). В таком списке с помощью специальных кнопок можно управлять отображением содержимого: разворачивать или сворачивать отдельные пункты. Для каждого узла дерева можно дополнительно (помимо текста) задать пиктограмму, которая может изменяться в зависимости от формы представления (в свернутом или развернутом виде).

Текстовые области. Текстовые области — различные элементы интерфейса, которые обеспечивают ввод, отображение и редактирование текстовых данных. Некоторые из них представляют собой комбинацию области обработки текста и элемента управления другого типа.

Текстовое поле (Text Box) — прямоугольная область, в которой фиксируется текстовый курсор и пользователь имеет возможность вводить и редактировать текст (рис. 3.32, *а*). Область может содержать одну или несколько строк. Для стандартного текстового поля поддерживаются операции вставки и удаления символов, а также выделение фрагментов. Текстовое поле может быть использовано только для отображения текста без возможности его редактирования, в этом случае в стандартном текстовом поле автоматически изменяется цвет фона.

Многострочное текстовое поле (Rich-Text Box) обеспечивает те же операции по работе с текстом, что и стандартное текстовое поле. Кроме этого, многострочное текстовое поле позволяет индивидуально настраивать шрифт для каждого символа, а также выбирать формат абзаца. Для такого поля реализованы функции печати содержимого и вставки объектов с использованием OLE-технологии (рис. 3.32, *б*).

Комбинированный список (Combo Box) представляет собой объединение текстового поля и списка в следующем подчинении: по мере того, как в текстовом поле происходит ввод текста,

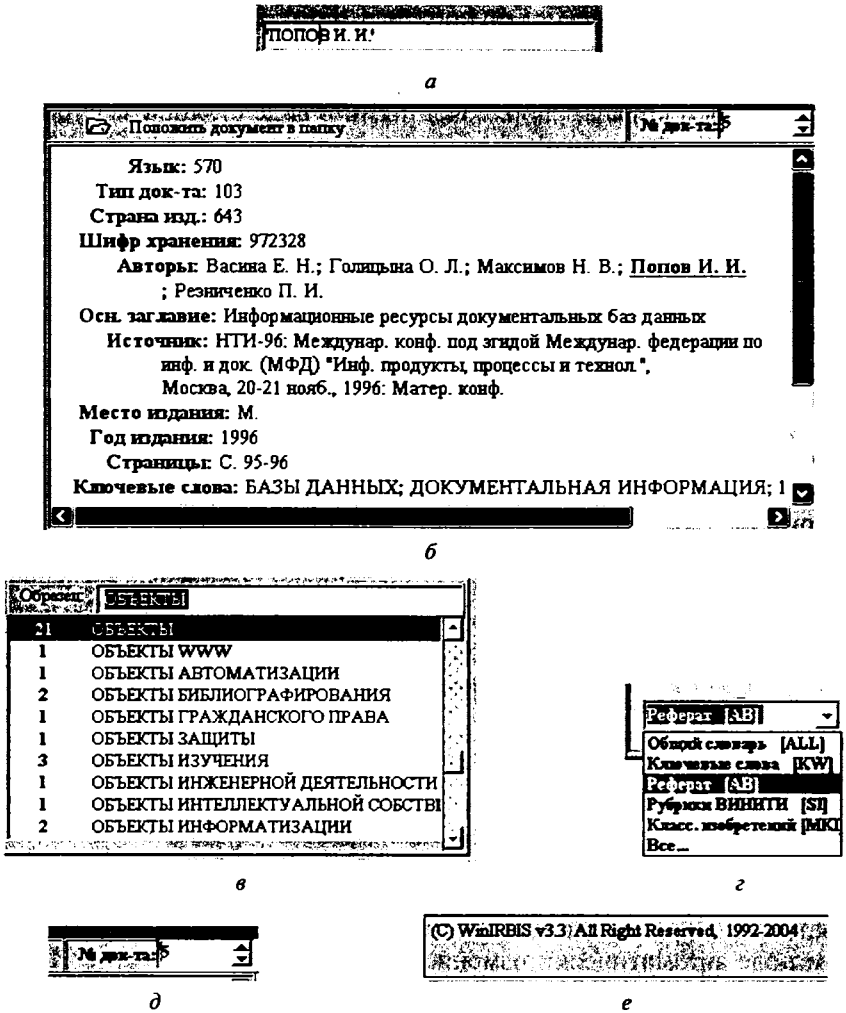


Рис. 3.32. Текстовые области:
а — текстовое поле; *б* — многострочное текстовое поле; *в* — комбинированный список; *г* — выпадающий комбинированный список; *д* — дискретное текстовое поле; *е* — статическая текстовая область

текущий указатель списка перемещается в соответствии с вводимыми символами. Когда же выбирается пункт в списке, он автоматически переносится в текстовое поле и становится текстом, который может быть отредактирован (рис. 3.32, *в*).

Выпадающий комбинированный список (Drop-down Combo Box) объединяет текстовое поле и выпадающий список и визуально отличается от выпадающего списка тем, что текстовое поле является интерактивным (т. е. может быть изменено). В открытом состоянии взаимосвязь текстового поля и списка осуществляется так же, как и в комбинированном списке (рис. 3.32, *г*).

Дискретное текстовое поле (Spin Box) представляет собой текстовое поле, в которое может быть введено только одно значение из ограниченного множества дискретных упорядоченных значений (рис. 3.32, *д*). Кнопки ▲ («вверх») и ▼ («вниз») позволяют автоматически увеличивать и уменьшать отображаемое значение.

Статическая текстовая область (Static Text Fields) служит для отображения информации, предназначенной только для чтения (рис. 3.32, *е*). Используется в основном для отображения комментирующих текстов. В поле отсутствует возможность выделения. Текст поля может быть изменен приложением в ходе работы.

Панели управления. К панелям управления относятся специальные компоненты графического интерфейса, предназначенные для создания функционально-ориентированных наборов элементов управления. К таким панелям относятся панели инструментов и строка состояний.

Панель инструментов (Toolbar) обычно объединяет элементы управления, обеспечивающие быстрый доступ к наиболее часто используемым командам или свойствам объектов (рис. 3.33, *а*, *б*, *в*). Панель инструментов создается в соответствии с определенным функциональным назначением (например, **Стандартная**, **Фоматирование**, **Автотекст** и т. п. в текстовом процессоре Microsoft Word).

Строка состояния (Status Bar) — специальная область внутри первичного окна (обычно в нижней его части), предназначенная для вывода информации о текущем состоянии объектов или процессов, представленных в окне (рис. 3.33, *г*).

Дополнительные элементы управления. Рассмотрим некоторые дополнительные элементы управления, которые могут быть полезны при разработке графических интерфейсов.

Заголовки столбцов (Column headings) используются для идентификации столбцов, содержащих формируемые приложением данные. При этом стандартный заголовок может быть разделен на произвольное количество частей. Такие заголовки при-

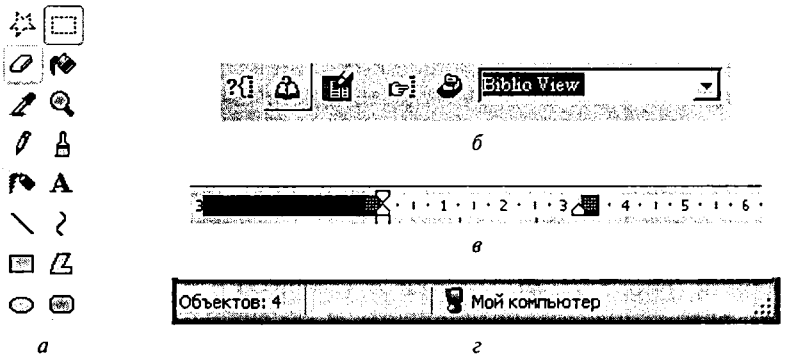


Рис. 3.33. Панели управления:
а, *б*, *в* — панели инструментов; *г* — строка состояния

меняются, например, в модифицируемых списках при выводе информации в табличном виде (рис. 3.34, *а*).

Этикетка вкладки (Tab) применяется для создания нескольких логически завершенных страниц (вкладок) в пределах одного окна. По форме и назначению этикетка вкладки аналогична разделителю в картотеке или записной книжке (рис. 3.34, *б*).

Полоса прокрутки (Scroll bar) представляет собой средство взаимодействия пользователя с ограниченной областью отображения для просмотра всей содержащейся в области информации (рис. 3.34, *в*).

Всплывающая подсказка (Tooltip Control) предназначена для отображения комментирующего текста в то время, когда пользователь устанавливает курсор на какой-либо элемент интерфейса. Подсказка появляется после короткой задержки и автоматически удаляется, если пользователь активизирует элемент или перемещает курсор (рис. 3.34, *г*).

Индикатор состояния процесса (Progress Indicator) используется для отображения хода выполнения длительной операции и не является интерактивным элементом (рис. 3.34, *д*).

Ползунковый регулятор (Slider) состоит из шкалы, определяющей диапазон возможных значений регулируемой величины, и индикатора, положение которого устанавливает и показывает текущее значение величины. Используется для установки величин, имеющих непрерывный диапазон значений, — яркость, громкость и т. п. (рис. 3.34, *е*)

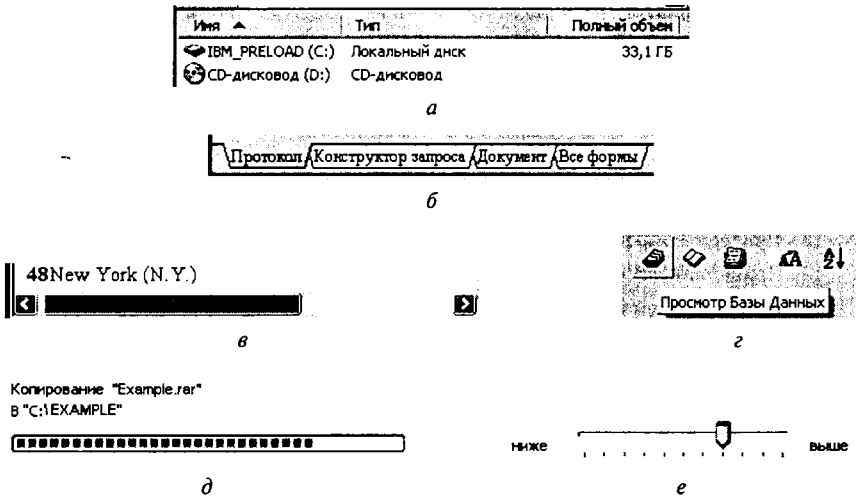


Рис. 3.34. Элементы управления:
а — заголовки столбцов; *б* — этикетки вкладок; *в* — полоса прокрутки; *г* — всплывающая подсказка; *д* — индикатор состояния процесса; *е* — ползунковый регулятор

Упражнения

1. Разработайте интерфейс (первичное окно, состав и размещение элементов управления, вторичные окна) для задачи ввода анкетных данных.

2. Приведите примеры алгоритмов, использующих фрагменты действий, которые могут быть оформлены в виде библиотечных подпрограмм.

Контрольные вопросы

1. Определите понятия «программное средство», «программный продукт».
2. Дайте определение жизненного цикла программных средств.
3. Перечислите и охарактеризуйте базовые этапы моделей ЖЦ.
4. Перечислите и охарактеризуйте стратегии конструирования программных средств с точки зрения моделей ЖЦ.
5. Определите инкрементную и эволюционную стратегии конструирования программных средств, приведите их характеристики и модели ЖЦ.

6. Охарактеризуйте разницу между компилирующей системой трансляции и интерпретирующей.
7. Охарактеризуйте признаки классификации систем программирования.
8. Охарактеризуйте структуру абстрактной системы программирования.
9. В чем преимущества использования библиотек подпрограмм?
10. Охарактеризуйте разницу между использованием библиотеки статического вызова и библиотеки динамического вызова.
11. Приведите характеристику CASE-средств.
12. Приведите классификацию CASE-средств.
13. Охарактеризуйте графический язык Диаграммы потоков данных (DFD).
14. Охарактеризуйте графический язык функциональной модели IDEF0.
15. Дайте характеристику графическому языку модели IDEF3.
16. Охарактеризуйте структуру UML.
17. Перечислите и охарактеризуйте сущности UML.
18. Определите содержание процесса тестирования.
19. Охарактеризуйте информационные потоки процесса тестирования.
20. Перечислите основные стратегии тестирования, приведите их характеристики, достоинства и недостатки.
21. Приведите основные характеристики первичного окна приложения.
22. Приведите основные характеристики вторичного окна приложения.

Глава 4
OBJECT PASCAL И ИНТЕГРИРОВАННАЯ СРЕДА РАЗРАБОТКИ ПРОГРАММ DELPHI

Целью главы является краткое описание основных возможностей и порядка разработки программ (далее по тексту — приложений) в интегрированной среде разработки Delphi.

Delphi позволяет создавать, редактировать, компилировать и тестировать приложение в единой среде программирования. В качестве базового языка в среде принят язык программирования Pascal, а именно — его объектно-ориентированное расширение Object Pascal.

На протяжении многих лет язык программирования Pascal довольно часто упоминается как в учебной, так и в научной литературе. Созданный специально с педагогическими целями, Pascal оказался крайне удачным не только в силу того, что ему несложно научиться, но и как основа обсуждения языков программирования вообще.

Предварительное описание языка программирования Pascal было опубликовано в 1968 г. его создателем — профессором кафедры вычислительной техники Швейцарского федерального института технологии Николасом Виртом (название свое язык получил в честь великого французского математика и механика Блеза Паскаля, создавшего в 1642 г. первую счетную машину). Это был язык, по духу продолжавший линию языков Алгол-60 и Алгол-W. Затем, после периода интенсивного развития, в 1970 г. заработал первый компилятор.

Растущий интерес к созданию компиляторов на других машинах привел к распространению языка. В начале 1980-х годов Pascal еще более упрочил свои позиции с появлением трансляторов MS-Pascal и Turbo-Pascal для персональных ЭВМ. С этого времени язык Pascal становится одним из наиболее широко используемых языков программирования для персональных ЭВМ не только как рабочий инструмент пользователя, но и как сред-

ство обучения программированию студентов высших учебных заведений.

Далее в развитии языка стала заметна тенденция его привязки к компьютеру IBM PC, стремление сделать его гибким и удобным инструментом. Следующий шаг усовершенствования — версия Object Pascal. Включив в себя понятие класса, эта версия языка поддерживает предыдущие версии Pascal в реализации фирмы Borland.

Глава содержит четыре раздела:

- описание языка программирования Object Pascal;
- описание основных возможностей среды Delphi;
- прядок разработки приложения;
- описание технологии работы с внешними источниками данных.

Описание языка включает общие сведения — синтаксис, типы данных, операции, операторы, объявление процедур и функций и т. п.

Во втором разделе дается краткое представление об основных интерфейсных объектах среды Delphi, позволяющих управлять процессом создания, сборки и отладки приложения. Описаны интерфейсные окна, связанные с созданием и редактированием проекта. Дана характеристика файлов, составляющих приложение, и описан порядок компиляции, сборки и отладки приложения. Далее рассматривается процесс разработки приложения с точки зрения взаимосвязи понятий «Форма», «Компонент», «Свойство компонента», «Событие» и «Процедура — обработчик события».

В третьем разделе описывается и поясняется на простых примерах содержательная составляющая понятий и последовательность действий при проектировании приложений.

В конце главы коротко охарактеризованы технологии и компоненты, обеспечивающие работу приложения с внешними источниками данных.

4.1. Язык программирования Object Pascal в Delphi

4.1.1. Лексика языка

При записи текстов программ на языке Object Pascal разрешается использовать прописные и строчные буквы латинского алфавита (A B C D E F G H I J K L N O P Q R S T U V W X Y Z

a b c d e f g h i j k l n o p q r s t u v w x y z), знак «_» («подчеркивание»), цифры (0 1 2 3 4 5 6 7 8 9) и ограничители.

Ограничители в тексте программы разделяют элементы фраз. Это могут быть знаки операций, скобки и другие служебные знаки или служебные слова, для которых не удалось подобрать служебные знаки.

Служебные знаки, выполняющие в языке определенные функции, делятся на знаки пунктуации, знаки операций и разделители.

Состав и назначение *знаков пунктуации* приведены в табл. 4.1. *Знаки операций* (+ - / * ^ = > >= < <=) используются при записи арифметических и логических выражений. *Разделителем* может служить знак пробела либо управляющий символ (коды от 0 до 31).

Таблица 4.1. **Назначение знаков пунктуации в Object Pascal**

Знаки	Назначение
(* *) { }	Скобки комментария. Текст, заключенный между скобками, поясняет алгоритм и не является его частью
[ ]	Задание индексов массива, размера строки, элементов множества
( )	Выделение части выражения, задание списков параметров
;	Отделение одного предложения программы от другого, разделение параметров (в части объявления)
:	Отделение переменной или константы от типа (в части объявлений), отделение метки от оператора, следующего за ней
,	Разделение элементов списка, параметров процедуры и функции при вызове
@	Обозначение адреса (переменной, константы, процедуры, функции, метода)
$	Признак числа в шестнадцатеричной системе, обозначение директивы компилятора
#	Обозначение символа по его коду
..	Разделение границ диапазона в типе-диапазоне
:=	Знак оператора присваивания
=	Отделение идентификатора типа (константы) от его описания (значения)
'	Апостроф — признак символа или строковой константы

Следующие *служебные (зарезервированные)* слова могут быть использованы только по своему специальному назначению:

and	**array**	**as**	**asm**
begin	**case**	**class**	**const**
constructor	**destructor**	**dispinterface**	**div**
do	**downto**	**else**	**end**
except	**exports**	**file**	**finalization**
finally	**for**	**function**	**goto**
if	**implementation**	**in**	**inherited**
initialization	**inline**	**interface**	**is**
label	**library**	**mod**	**nil**
not	**object**	**of**	**or**
out	**packed**	**procedure**	**program**
property	**raise**	**record**	**repeat**
resourcestring	**set**	**shl**	**shr**
string	**then**	**threadvar**	**to**
try	**type**	**unit**	**until**
uses	**var**	**while**	**with**
xor			

Слова **private**, **protected**, **public**, **published** и **automated** зарезервированы для объявления объектного типа данных (класса) и воспринимаются компилятором как директивы. Слова **at** и **on** также имеют специальное назначение.

Для именования различных алгоритмических объектов служат языковые конструкции, называемые *идентификаторами*. Идентификатор определяется как последовательность букв и цифр, начинающаяся с буквы.

При записи текста программы нет разницы между строчными и прописными буквами.

4.1.2. Переменные и константы, базовые типы данных

Для объявления переменных и констант в программе выделены особые синтаксические разделы. Раздел объявления констант начинается со служебного слова `const` и содержит перечень всех используемых в программе констант, например:

```
const
       Radius = 4;
```

Раздел объявления переменных начинается со служебного слова var и содержит описание всех переменных:

```
var
     Radius: Integer
```

Типы данных в языке Object Pascal можно разделить на предопределенные в языке (встроенные) типы и типы, определяемые программистом (пользовательские). При этом встроенные типы используются при объявлении переменных и пользовательских типов. Все пользовательские типы должны быть объявлены в разделе, идентифицированном ключевым словом type, следующим образом:

<Идентификатор типа> = <Описание типа>;

Встроенные типы, в свою очередь, подразделяются на простые (базовые) и структурированные.

К базовым типам относятся:

- целый;
- вещественный;
- логический;
- символьный;
- перечисляемый;
- тип-диапазон.

Целый тип данных

Целый тип данных представлен множеством типов: Shortint, Smallint, Integer, Longint, Int64, Byte, Word, Longword, Cardinal (табл. 4.2)

Например, для переменных X, Y, Z, описанных в разделе объявления переменных, как

```
var
     X: Byte;
     Y: Smallint;
     Z: Word;
```

Операторы X:=2000; Y:=-40000; Z:=-2 будут некорректными, так как переменная X не может принимать значение, большее чем 255, Y не может быть меньше чем –32 768, Z должно быть положительным.

Таблица 4.2. **Базовые типы данных Object Pascal**

Название типа	Область изменения данных	Занимаемый размер в байтах
Shortint	–128..+127	1
Smallint	–32 768..+32 767	2
Integer	$-2^{31}..+2^{31}-1$	4
Longint	$-2^{31}..+2^{31}-1$	4
Int64	$-2^{63}..+2^{63}-1$	8
Byte	0..+255	1
Word	$0..+2^{16}-1$	2
Longword	0..+4 294 967 295	4
Cardinal	0..+4 294 967 295	4
Boolean	True..False	1
ByteBool	True..False	1
WordBool	True..False	2
LongBool	True..False	4
Char	Один символ	1
ANSIChar1	Один символ ANSI	1
WideChar1	Один символ Unicode	2
Real	±5.0 · 10—324.. ±1.7 · 10 308	8
Real48	±2.9 · 10—39.. ±1.7 · 1038	6
Single	±1.5 · 10—45.. ±3.4 · 1038	4
Double	±5.0 · 10—324.. ±1.7 · 10 308	8
Extended	±3.6 · 10—4951..1.1 · 104 932	10
Comp	–263+1..+263 –1	8
Currency	–922 337 203 685 477.5808 ..+922 337 203 685 477.5807	8

Следует отметить, что максимальную производительность обеспечивают типы Integer и Cardinal. Типы Integer и Longint, Cardinal и LongWord в таблице имеют одинаковые размеры, но в старших версиях языка размеры типов с префиксом Long могут быть увеличены.

Логический тип данных

Данные в стандарте языка могут принимать одно из двух значений — `True` или `False`. Переменная или константа логического типа занимает 1 байт, в который записывается 1, если переменная или константа имеет значение True, и 0 в противном случае.

Помимо стандартного логического типа (`Boolean`) в реализации языка Object Pascal добавлено еще три логических типа (для совместимости с другими языками программирования и со средой Windows): `ByteBool`, `WordBool`, `LongBool` (табл. 4.2). Основные отличия этих типов от стандартного заключаются, во-первых, в фактическом размере (в байтах), а во-вторых — в величине, соответствующей значению `True`. Для всех логических типов значению `False` соответствует число 0, записанное в соответствующее количество байтов. Значению же `True` в случае стандартного типа соответствует только 1, записанная в байт, а в случае других логических типов — любое число, отличное от 1.

Между значениями `True` и `False` имеют место следующие соотношения:

Boolean:

```
False < True
Ord(False) = 0
Ord(True) = 1
Succ(False) = True
Pred(True) = False
```

ByteBool, WordBool, LongBool:

```
False <> True
Ord(False) = 0
Ord(True) <> 0
Succ(False) = True
Pred(False) = True
```

В отличие от некоторых других языков программирования в Object Pascal нет возможности трактовать целое значение как величину логического типа, т. е. использование, например, переменной `X` целого типа в выражении

```
While X do ...
```

вызовет ошибку компиляции.

Символьный тип данных

Данные стандартного символьного типа `Char` представляют собой символы раскладки ASCII. Переменная или константа символьного типа занимает 1 байт памяти. В соответствии с синтаксисом языка значение символа заключается в одинарные

кавычки: 'P', 'a', 's', 'c', 'a', 'l'. Включение в состав языка дополнительных типов — ANSIChar и WideChar — связано с необходимостью представления двух различных множеств символов: ANSI и Unicode (см. табл. 4.2).

При вызове функции Ord(Ch), где Ch — значение типа Char, возвращается порядковый номер Ch. Любое значение символьного типа можно получить с помощью стандартной функции Chr(X), где X — значение типа byte, или используя знак «#» перед числом, например #30 = Chr(30).

Для следующих объявлений переменных

```
var
 C:Char;
 B:Byte;
```

справедливы операторы:

```
C := #3;
B := Ord(C);
B := 125;
C := Chr(B);
```

Перечисляемый тип данных

Перечисляемый тип данных задается упорядоченным набором идентификаторов, с которыми могут совпадать значения переменной (или константы) этого типа. Список идентификаторов указывается в круглых скобках, а сами идентификаторы разделяются запятыми:

```
<Идентификатор типа> = (<Идентификатор>
                  [,<Идентификатор>...]);
```

Упорядочение наборов выполняется в соответствии с последовательностью, в которой перечисляются идентификаторы. Порядковый номер перечисляемой константы определяется ее позицией в списке идентификаторов при объявлении. Первый идентификатор в списке имеет порядковый номер 0, второй — порядковый номер 1 и т. д. Один и тот же идентификатор можно использовать в объявлении только одного перечисляемого типа.

Примеры перечисляемого типа:

```
type
   Color = (Red, Green, Blue, Black);
   Operator = (Plus, Minus, Multiply, Divide);
```

Порядковые типы данных

Типы данных логический, символьный, целый и перечисляемый относятся к *порядковым типам*, т. е. представляют собой множества значений, для которых определен порядок следования.

Для величин порядкового типа существуют стандартные процедуры и функции, позволяющие выполнить ряд действий. Перечень таких процедур и функций приведен в табл. 4.3.

Таблица 4.3. **Стандартные процедуры и функции для порядковых типов данных**

Название	Действие	Тип аргумента (параметра)	Тип результата	Примеры

`Ord(X)`	Возвращает целое число, которое показывает, какое положение занимает значение X по отношению к другим значениям	Логический, символьный, целый, перечисляемый	`Longint`	`Ord(false)=0:`
				`Ord('1')=34;`
				`Ord(Club)=0`

`Pred(X)`	Возвращает предыдущее значение X. При применении к первому элементу возникает ошибка	Логический, символьный, целый, перечисляемый	Тот же, что и у аргумента	`Pred(true)=false;`
				`Pred(0)=-1;`
				`Pred(Minus)=Plus`
				`Pred(Plus)-error!`

`Succ(X)`	Возвращает последующее значение X. При применении к последнему элементу возникает ошибка	Логический, символьный, целый, перечисляемый	Тот же, что и у аргумента	`Succ(false)=true;`
				`Succ('1')='2';`
				`Succ(0)=1;`

`Odd(X)`	Проверка аргумента на нечетность: возвращает `true`, если аргумент нечетный, и `false`, если аргумент четный	`Longint`	`Boolean`	`Odd(3)=true;` `Odd(6)=false;`
`Inc(X[,N])`	Процедура. Увеличивает значение переменной X на 1 (если второй параметр отсутствует) или на N	Тип X — целый, логический, символьный, перечисляемый; тип N — `Longint`	Тот же, что и у аргумента X	`Inc(X)` аналогично `X:=X+1`, если X — целое; `Succ(X)`, если X — перечисляемое

Окончание табл. 4.3

Название	Действие	Тип аргумента (параметра)	Тип результата	Примеры
`Dec(X[,N])`	Процедура. Уменьшает значение переменной `X` на 1 (если второй параметр отсутствует) или на `N`	Тип `X` — целый, логический, символьный, перечисляемый; тип `N` — `Longint`	Тот же, что и у аргумента `X`	`Dec(X)` аналогично `X:=X-1`, если `X` — целое; `-Pred (X)`, если `X` — перечисляемое
`High(X)`	Возвращает максимально возможное значение	Логический, символьный, целый, перечисляемый. Может быть идентификатором порядкового типа	Тот же, что и у аргумента	`High(byte) = 255` `High(B) = true`, если `B` имеет тип `boolean`
`Low(X)`	Возвращает мимимально возможное значение	Логический, символьный, целый, перечисляемый. Может быть идентификатором порядкового типа	Тот же, что и у аргумента	`Low(byte) = 0`

Тип-диапазон

Тип-диапазон представляет собой диапазон значений из порядкового типа, называемого главным типом. Определение типа-диапазона задает все значения из главного типа, находящиеся между наименьшим и наибольшим значением, включая сами границы, и имеет следующий синтаксис:

```
<Имя типа-диапазона> = <Константа-мин. значение>..
                       < Константа-макс. значение>;
```

Обе константы должны быть одного порядкового типа, при этом минимальное значение не должно превышать максимального.

Приведем примеры типов-диапазонов:

```
0 .. 999
-100 .. 100
Red .. Blue
```

Переменная типа диапазон имеет все свойства переменных главного типа, а ее значение на этапе выполнения должно принадлежать указанному интервалу.

Для задания границ диапазона разрешается использовать константные выражения, но в таком случае может возникнуть синтаксическая неопределенность. Рассмотрим следующие объявления:

```
const
   X = 5;
   Y = 10;
type
   Operator = (Plus, Minus, Multiply, Divide);
   Height = (X - Y) * 2  ..  (X + Y) * 2;
```

В соответствии с правилами синтаксиса языка любое определение типа, начинающееся с круглой скобки, воспринимается, как определение перечисляемого типа (см. определение типа `Operator`). Однако целью объявления `Height` является задание типа-диапазона. Решением этой проблемы является либо преобразование первого выражения, задающего минимальное значение, так, чтобы оно не начиналось с круглой скобки, либо задание другой константы, равной значению этого выражения, и затем использование этой константы в определении типа, например:

```
type
   Height = 2 * (X - Y)       (X + Y) * 2;
```

Вещественный тип данных

Вещественный тип в стандарте языка Pascal называется `Real`. Помимо типа `Real` в реализациях Object Pascal определены еще шесть стандартных вещественных типов: `Real48`, `Single`, `Double`, `Extended`, `Comp`, `Currency` (см. табл. 4.2).

Каждый тип характеризуется своей областью изменения возможных значений.

Тип `Comp` фактически является целым типом расширенного диапазона, но при этом на считается порядковым (т. е. к переменным типа `Comp` нельзя применять процедуры и функции, определенные только для порядковых типов — `Inc`, `Dec` и т. п.).

Выбор конкретного типа для переменной связан с требуемой точностью вычислений.

Для переменных вещественного типа определены две функции, позволяющие преобразовать переменную вещественного типа в переменную целого типа: `Round(X)` — округление веще-

ственного числа до целого и `Trunc(X)` — выделение целой части числа. В качестве аргументов функций выступают значения вещественного типа, а результат принадлежит целому типу, например:

```
Round(3.456) = 3; Round(5.678) = 6;
Trunc(3.456) = 3; Trunc(5.678) = 5;
```

Типизированные константы

Типизированные константы можно использовать точно так же, как переменные того же самого типа, и они могут появляться в левой части оператора присваивания. Отметим, что типизированные константы задаются только один раз — в начале выполнения программы. Таким образом, при каждом новом входе в процедуру или функцию локально объявленные типизированные константы заново не инициализируются.

Синтаксис объявления типизированной константы следующий:

```
<Идентификатор>:<Тип данных> = <Значение>;
```

Константы простого типа. Объявление типизированной константы простого типа содержит в своем описании указание на простой тип данных (например, `Integer`, `Real` или `Char`):

```
const
   Maximum   : Integer = 999;
   Factor    : Real    = -0.1;
   Breakchar : Char    = #3;
```

Поскольку типизированная константа фактически представляет собой переменную с константным значением, она не может заменять обычную константу. Например, она не может использоваться в объявлении других констант или типов:

```
const
   Min : Integer = 1;
   Max : Integer = 1000;
type
   Vector = array [Min..Max] of Integer;
```

Объявление `Vector` является недопустимым, поскольку `Min` и `Max` являются типизированными константами.

4.1.3. Структурированные типы данных

В языке Object Pascal определены следующие стандартные структурированные типы данных:

- массив;
- строка;
- запись;
- файл;
- множество;
- класс.

По умолчанию данные в структурированных типах выравниваются по границам машинных слов или двойных слов, что обеспечивает быстрый доступ к данным. Однако такой подход к размещению данных иногда сказывается на пространстве хранения: не обеспечивается компактное хранение данных. Для устранения этого недостатка в Object Pascal существует ключевое слово `packed` (*упакованный*), которое используется при объявлении структурированных типов (перед именем типа) и обеспечивает компактное хранение (т. е. данные будут занимать ровно столько памяти, сколько необходимо для непрерывного хранения значений типа). Например:

```
type TBytes = packed array [1..100] of Byte;
   TPackedStr = packed array [1..128] of Char;
```

Следует, однако, помнить о том, что использование упакованных типов замедляет доступ к данным.

Структуры типа `TPackedStr` называются *упакованными строками* и совместимы со строковыми типами.

Массив

Общий синтаксис объявления типа N-мерный массив:

```
<имя>=array [<тип_индекса1>,<тип_индекса2>,...,<тип_индексаN>] of
      <тип_элемента>
```

Индексы массива должны принадлежать типу-диапазону, т. е. принимать множество последовательных значений порядкового типа. Элемент массива может принадлежать любому типу данных (в том числе и типу массив), например:

```
type
    Real_array = array [1..365] of Real;
```

Задание значений массиву-константе. При определении массива-константы значения элементов массива указываются в круглых скобках и разделяются запятыми. Если массив многомерный, то внешние скобки соответствуют самому левому индексу, вложенные в них — следующему, и т. д.

Например, для массивов, типы которых определяется предложениями:

```
type
   Vect_array = array [1..7] of Integer;
   Matr_array = array [1..3, 1..4] of Integer;
```

Могут быть заданы константы C1 и C2:

```
const
   C1: Vect_array = (1,3,5,7,9,11,13);
   C2: Matr_array = ((1,3,5,7),(2,4,6,8),(9,11,13,15));
```

Константа C2 соответствует следующей структуре:

1	3	5	7
2	4	6	8
9	11	13	15

Операции над массивами. Для одномерных массивов символов можно использовать *операции сравнения*, даже если массивы не идентичных типов и имеют различный размер, например, для объявленных следующим образом массивов:

```
var
A : array [1..15] of Char;
B : array [1..10] of Char;
```

можно написать условный оператор вида

```
if A>B   then ShowMessage(A)
         else ShowMessage(B);
```

Одному массиву можно присвоить значение другого, но только если они идентичных типов. Например, если заданы массивы:

```
var
A1, A2 : array [1..15] of Real;
B : array [1..15] of Real;
```

то оператор присваивания допустим только между массивами A1 и A2 (A1 := A2), несмотря на то, что размеры и типы элементов совпадают у всех трех массивов.

Стандартные функции. В табл. 4.4 приведены стандартные функции для работы с массивами. Функции Length, High и Low определены для любых массивов. Остальные функции применимы только для массивов целых и вещественных чисел и определены в Object Pascal в модуле Math (для использования функций модуль должен быть подключен в разделе Uses).

Таблица 4.4. **Стандартные функции для работы с массивами**

Функция	Описание	Тип массива	Тип результата
Length(Data)	Число элементов массива	Любой	Целый
High(Data)	Наибольшее значение индекса	Любой	Тип индекса
Low(Data)	Наименьшее значение индекса	Любой	Тип индекса
MaxIntValue(Data)	Возвращает значение максимального элемента	Целых чисел	Целый
MaxValue(Data)	Возвращает значение максимального элемента	Вещественных чисел	Вещественный
Mean(Data)	Возвращает среднее арифметическое элементов массива	Вещественных чисел	Вещественный
MinIntValue(Data)	Возвращает значение минимального элемента	Целых чисел	Целый
MinValue(Data)	Возвращает значение минимального элемента	Вещественных чисел	Вещественный
Sum(Data)	Возвращает сумму элементов массива	Вещественных чисел	Вещественный
SumInt(Data)	Возвращает сумму элементов массива	Целых чисел	Целый
SumOfSquares(Data)	Возвращает сумму квадратов элементов массива	Вещественных чисел	Вещественный

Динамические массивы. Динамические массивы, в отличие от обычных статических, характеризуются тем, что для них заранее не объявляется длина — число элементов. Такие массивы удобно использовать в алгоритмах, где объемы обрабатываемых данных заранее неизвестны и определяются интерактивно в процессе выполнения.

Объявление одномерного динамического массива содержит только его имя и тип элементов:

```
<имя> array of <базовый тип>
```

Например, объявление динамического массива байтов:

```
var Ar: array of Byte;
```

Многомерный динамический массив объявляется как динамический массив динамических массивов и т. д. Например, двумерный динамический массив байтов:

```
var Ar2: array of array of Byte;
```

При объявлении динамического массива память под него не отводится, поэтому перед использованием массива его необходимо разместить и задать размер с помощью процедуры `SetLength`. Процедура имеет два параметра — имя массива и количество размещаемых элементов, например:

```
SetLength(Ar, 100);
```

выделяет для массива Ar место в памяти размером 100 байт и присваивает нулевые значения всем размещенным элементам. Повторное применение к массиву функции приведет к изменению длины массива: если новый размер больше предыдущего, то массив будет перераспределен, все его старые значения элементов будут сохранены, а новые — проинициализированы значением 0; если новый размер меньше предыдущего — массив будет усечен до новой длины.

Для размещения многомерного массива можно использовать функцию `SetLength` с количеством параметров, необходимым для задания всех размерностей, например:

```
SetLength(Ar2, 100, 100);
```

размещает матрицу байтов размерностью 100 на 100.

Индексы динамических массивов — обязательно целые числа. Первый элемент динамического массива всегда имеет индекс 0.

После размещения динамического массива в памяти к нему можно применять стандартные функции. Функция `Length` возвращает текущую длину массива; функция `High` — наибольшее значение индекса (результат будет всегда на единицу меньше,

чем результат функции Length); функция Low — наименьшее значение индекса (для динамического массива всегда равно 0).

Переменная динамического массива является указателем на начало массива и имеет значение nil, если место под массив еще не выделено. Однако это не означает, что с переменной можно работать так же, как с переменными типа указатель: ее нельзя передать как параметр в процедуры new и Dispose, нельзя применить операцию разыменования (^).

Удалить динамический массив из памяти можно одним из следующих способов:

- присвоить ему значение **nil**: `Ar := nil;`
- использовать специальную функцию `Finalize(Ar);`
- задать нулевую длину `SetLength(Ar, 0).`

При применении операций сравнения к динамическим массивам (правила применения те же, что и для статических массивов) сравниваются только сами указатели, поэтому, если выражение `Ar1 = Ar2` имеет значение `true`, переменные Ar1 и Ar2 указывают на один и тот же массив, размещенный в памяти.

Строка символов

Тип строка символов (string) определяет последовательность символов произвольной длины, заключенную в апострофы (одинарные кавычки). Строка символов, ничего не содержащая между апострофами, называется пустой строкой. Два последовательных апострофа в строке символов обозначают один символ-апостроф.

Строку можно рассматривать как массив символов, однако, в связи с некоторыми особенностями использования строк по сравнению со стандартными массивами, символьный массив выделен в отдельный (строковый) тип данных.

В целях преодоления ограничений на размер стандартных строк в 32-разрядных реализациях языка Pascal (в визуальной среде разработки приложений Delphi) введена поддержка *длинных строк*. Строковые типы, которые могут быть объявлены в Object Pascal, приведены в табл. 4.5.

Тип string, как видно из таблицы, может интерпретироваться (в зависимости от директивы компилятора) и как «короткая», и как «длинная» строка. По умолчанию тип string совпадает с типом AnsiString.

Таблица 4.5. **Строковые типы данных**

Тип	Максимальная длина, символов	Описание
`ShortString`	255	Соответствует стандартному типу `string`. Каждый элемент строки имеет стандартный символьный тип
`string`	255 или $\approx 2^{31}$	Соответствует стандартному типу `string` или «длинная» строка переменной длины
`AnsiString`	$\approx 2^{31}$	«Длинная» строка переменной длины. Память выделяется динамически. Каждый элемент строки имеет стандартный символьный тип. Ограничивается нулевым значением (`#0`)
`WideString`	$\approx 2^{30}$	«Длинная» строка переменной длины. Память выделяется динамически. Каждый элемент строки представлен символом стандарта UniCode. Ограничивается нулевым значением

Служебное слово `string` является общим при объявлении как коротких, так и длинных строк. Например, объявление

```
var
    S: string;
```

создает переменную типа `AnsiString`, если компилятор поддерживает размещение длинных строк, и переменную типа `ShortString` в противном случае.

Различия между «короткими» и «длинными» строками заключаются в следующем:

1. «Короткая» строка размещается в памяти при объявлении переменной соответствующего типа.

Переменная типа «длинная» строка представляет собой указатель на динамически выделяемую под хранение строки область памяти и, следовательно, перед использованием строка должна быть размещена с помощью специальной процедуры `SetLength`. Таким образом, может быть несколько переменных, ссылающихся на одну и ту же физическую строку (т. е. оператор присваивания `str1 := str2` на самом деле копирует не саму строку, а указатель), что позволяет экономить ресурсы памяти. Если длина строки равна 0, то указатель на строку имеет значение `nil`. Если же строка не пустая, то в выделенной под нее памяти хранится сама строка, ее длина и количество ссылок на строку

(число переменных, ссылающихся на эту строку). Если в строке производятся изменения (через некоторую переменную), то она копируется (появляется ее новый экземпляр) только в том случае, если число ссылок на нее больше 1. В случае копирования число ссылок в текущей строке уменьшается на 1, а в новой — увеличивается на 1.

2. У «коротких» строк атрибут длины строки содержится в символе с порядковым номером 0 и представляет собой размер строки, числовое значение которого определяется как `ord(string[0])`. В стандарте языка Pascal строковый тип имеет фиксированный или динамический атрибут длины, но в любом случае длина строки не может превышать 255 символов. Фиксированный атрибут длины задается в квадратных скобках после слова `shortstring` (или `string`) при объявлении типа. Например, строка, объявленная как

```
var
   MyString : string[20];
```

может иметь длину не более 20 символов. В памяти при этом размещается массив длиной 21 символ (нулевой символ — под длину строки). Тип «короткая» строка, объявленный без атрибута длины, имеет установленный по умолчанию атрибут длины, равный 255. Текущее значение атрибута длины можно получить с помощью стандартной функции `Length`. При этом возвращается число символов в строке, не считая нулевого.

У «длинных» строк атрибут длины доступен только с помощью стандартной функции `Length`. Процедуры `SetLength` назначают новую длину строки.

3. «Длинные» строки ограничиваются нулевым терминатором, т. е. завершаются байтом с нулевым значением, что обеспечивает их совместимость со строками языка C, C++.

К символам в строке любого типа можно иметь доступ как к компонентам массива, например, для объявленной строки `MyString`:

```
var
   MyString : string;
```

можно программировать следующие действия:

```
MyString[1]:='H'; MyString[2]:='E'; MyString[3]:='L';
MyString[4]:='L'; MyString[5]:='O';
```

Эта последовательность действий будет аналогична оператору

```
MyString:='HELLO';
```

если в атрибуте длины строки установить значение 5.

В Object Pascal разрешено вставлять в строку символов управляющие коды. Символ «#» с целой константой без знака в диапазоне от 0 до 255 обозначает соответствующий этому значению символ в коде ASCII. Между символом «#» и целой константой не должно быть никаких разделителей. Если в строку символов входит несколько управляющих символов, то между ними также не должно быть разделителей.

Приведем несколько примеров строк символов:

```
'Object Pascal'
'You''ll see!!!'
#13#10'Line 1'#13'Line 2'#7#7'The End!'#7#7
```

Константы строкового типа. Объявление типизированной константы строкового типа содержит максимальную длину (в случае «коротких строк») и ее начальное значение, при этом нулевой символ в конце «длинных» строк добавляется автоматически:

```
const
   HeadText : Shortstring[6] = 'Раздел';
   Newline  : string = #13#10;
   HeadProg : AnsiString = 'Текст программы';
```

Операции отношения для строковых типов. Любые два значения строковых данных можно сравнить, поскольку все значения строковых данных совместимы.

Операции отношения (=, <>, <, >, <=, >=) применяются для сравнения строк в соответствии с порядком расширенного набора символов применяемого кода (ASCII, ANSI, Unicod). Значение отношения между любыми двумя строками устанавливается согласно отношению порядка между значениями символов в одинаковых позициях. Все операции отношения учитывают регистр.

В двух строках разной длины каждый символ более длинной строки, для которого нет соответствующего символа в более короткой строке, принимает значение «больше», например: `'xs'` больше, чем `'x'`, но `'xxx'` меньше, чем `'xy'`.

Пустые строки могут быть равны только другим пустым строкам, и они являются строками с наименьшим значением.

Значения символьного типа совместимы со значениями строкового типа, и при их сравнении символьное значение обрабатывается как строковое значение длиной 1.

Стандартные процедуры и функции для работы со строками. Для работы с переменными строкового типа определены стандартные процедуры и функции. Некоторые из них (наиболее часто используемые) приведены в табл. 4.6 (в столбце «Статус» значение F соответствует функции, P — процедуре). Функции, имена которых начинаются с префикса `Ansi`, применимы к русским текстам.

Таблица 4.6. **Стандартные процедуры и функции для работы со строками**

Имя	Статус	Назначение	Аргументы (параметры)	Результат
`CompareStr(S1,S2)` `AnsiCompareStr(S1,S2)`	F	Сравнение двух строк с учетом регистра	`S1, S2` — строки	< 0, если `S1 < S2`; = 0, если `S1 = S2`; > 0, если `S1 > S2`.
`CompareText(S1,S2)` `AnsiCompareText(S1,S2)`	F	Сравнение двух строк без учета регистра	`S1, S2` — строки	< 0, если `S1 < S2`; = 0, если `S1 = S2`; > 0, если `S1 > S2`.
`Concat(S1,S2,...,Sn)`	F	Конкатенация (сложение) двух или более строк	`S1, S2, ..., Sn` — строки	Строка, равная `S1 + S2 + ... + Sn`
`Copy(S,Index,Count)`	F	Выделение подстроки	`S` — строка; `Index, Count` — целые	Часть строки `S`, начиная с позиции `Index`, длиной `Count`
`Delete(S,Index,Count)`	P	Удаление части строки	`S` — строка; `Index, Count` — целые	Новое значение строки `S` — без фрагмента, начиная с позиции `Index`, длиной `Count`
`Insert(Source,S,Index)`	P	Добавление подстроки в строку	`Source, S` — строки; `Index` — целое	Новое значение строки `S` после добавления в нее `Source`, начиная с позиции `Index`
`Length(S)`	F	Вычисление длины строки	`S` — строка	Целое значение длины строки
`LowerCase(S)` `AnsiLowerCase(S)`	F	Преобразование строки к нижнему регистру	`S` — строка	Новое значение строки `S` — все символы строчные

Окончание табл. 4.6

Имя	Статус	Назначение	Аргументы (параметры)	Результат
`Pos(Substr,S)`	F	Вычисляет позицию начала подстроки в строке	`Substr`, `S` — строки	=0, если `Substr` не содержится в `S`; целое >0 — позиция `Substr` в `S`
`SetLength(S,NewLength)`	P	Назначает новую длину строки	`S` — строка; `NewLength` — целое	Строка `S` с новым значением длины
`Str(X,S)`	P	Преобразует числовой тип в строковый	`X` — целое или вещественное; `S` — строка	Строковая запись (в строке `S`) числа `X`
`StringOfChar(Ch,Count)`	F	Формирует строку из последовательности символов	`Ch` — символ; `Count` — целое	Строка символов `Ch` длиной `Count`
`Trim(S)`	F	Удаляет пробелы и управляющие символы в начале и в конце строки	`S` — строка	Новая строка
`TrimLeft(S)`	F	Удаляет пробелы и управляющие символы в начале строки	`S` — строка	Новая строка
`TrimRight(S)`	F	Удаляет пробелы и управляющие символы в конце строки	`S` — строка	Новая строка
`UpperCase(S)` `AnsiUpperCase(S)`	F	Преобразование строки к верхнему регистру	`S` — строка	Новое значение строки `S` — все символы строчные
`Val(S,V,Code)`	P	Преобразование строкового типа в числовой	`S` — строка; `V` — целое или вещественное; `Code` — целое	Если `Code` =0, то целое или вещественное значение `V`, соответствующее строковой записи `S`. Если `Code` >0, то преобразование невозможно и значение `Code` указывает позицию ошибочного символа

Примеры использования строковых процедур и функций:

- выделение первого слова в предложении (разделитель слов — знак «пробел»):

```
S_Sentence := TrimLeft(S_Sentence); // Удаление пробелов
                                    // в начале строки
i := Pos(' ', S_Sentence); // Определение позиции первого
                           // пробела в предложении S_Sentence
if i > 0 then S_Word := Copy(S_Sentence, 1, i-1)
else S_Word := S_Sentence;
     // В строковой переменной S_Word — значение первого слова
```

- удаление из строки всех цифр:

```
L := Length(S_Sentence);     // Вычисление длины строки
i := 1;
while i <= L do
  if S_Sentence[i] in ['0'.. '9'] then
    begin Delete(S_Sentence, i, 1);
          dec(L)  // Уменьшение длины строки на 1 в случае
                  // удаления цифры
    end
  else inc(i);
```

- подсчет количества букв 'W' в строке (независимо от регистра):

```
N_w := 0;    // Обнуление счетчика букв
S_Sentence := LowerCase(S_Sentence);
            // Преобразование строки к нижнему регистру
L := Length(S_Sentence); // Подсчет длины строки
for I := 1 to L do
  if S_Sentence[i] = 'w' then inc(N_w);// Увеличение счетчика
                                       // букв на 1
```

Строка PChar

Для совместимости с другими языками программирования (например, C) и средой Windows в Object Pascal существует еще один вид строк — строки, оканчивающиеся нулевым байтом (#0). Этим строкам соответствует стандартный тип `PChar`. Фактически тип `PChar` эквивалентен массиву символов от 0 до N, где N — количество символов, не считая завершающего нулевого байта:

```
PChar = array [0..N] of Char;
```

В отличие от типа `string` символ с индексом `0` в `PChar`-строке является первым, а символ с индексом `N` — завершающим с кодом 0.

Для преобразования типов `string` и `PChar` в Object Pascal можно применять следующие действия: пусть объявлены переменные

```
var
   S_String: string;
   S_PChar: PChar;
```

Тогда справедливы следующие операторы присваивания:

```
S_String:= string(S_PChar);
S_PChar := PChar(S_String);
```

В Object Pascal для работы с переменными типа `PChar` определены стандартные функции (функции, имена которых начинаются с префикса `Ansi`, применимы к русским текстам). Некоторые из них приведены в табл. 4.7.

Таблица 4.7. **Стандартные функции для работы с `PChar`-строками**

Имя	Назначение	Аргументы (параметры)	Результат
`StrCat(Dest,Source)`	Объединение двух строк (добавление к PChar-строке `Dest` PChar-строки `Source`)	`Dest`, `Source` — PChar-строки	Новая PChar-строка
`StrComp(Str1,Str2)` `AnsiStrComp(Str1, Str2)`	Сравнение двух PChar-строк	`Str1`, `Str2` — PChar-строки	Целое число: <0, если `Str1<Str2`; =0, если `Str1=Str2`; >0, если `Str1>Str2`;
`StrCopy(Dest,Source)`	Копирование одной PChar-строки (`Source`) в другую (`Dest`)	`Dest`, `Source` — PChar-строки	PChar-строка `Dest`
`StrECopy(Dest, Source)`	Копирование одной PChar-строки (`Source`) в другую (`Dest`). Возвращается указатель на нулевой байт новой строки	`Dest`, `Source` — PChar-строки	PChar-строка, начинающаяся с нулевого байта `Dest`
`StrEnd(Source)`	Возвращается указатель на PChar-строку	`Source` — PChar-строка	PChar-строка, начинающаяся с нулевого байта `Source`

Продолжение табл. 4.7

Имя	Назначение	Аргументы (параметры)	Результат
`StrLen(Source)`	Возвращается количество символов в PChar-строке (исключая нулевой байт)	`Source` — PChar-строка	Целое число — количество символов в строке
`StrLower(Source)` `AnsiStrLower(Source)`	Преобразование символов PChar-строки к нижнему регистру (все строчные)	`Source` — PChar-строка	Преобразованная PChar-строка
`StrMove(Dest, Source, Count)`	Копирование заданного количества символов (`Count`) из одной PChar-строки (`Source`) в другую (`Dest`)	`Dest`, `Source` — PChar-строки; `Count` — целое	PChar-строка `Dest`
`StrPCopy(Dest, Source)`	Копирование строки `Source` типа `string` в PChar-строку `Dest`	`Dest` — PChar-строка; `Source` — строка типа `string`	PChar-строка `Dest`
`StrPos(Str1,Str2)` `AnsiStrPos(Str1, Str2)`	Возвращается указатель на начало PChar-строки `Str2` внутри PChar-строки `Str1`	`Str1`, `Str2` — PChar-строки	PChar-строка `Str2` внутри `Str1`; `nil` — если вхождение не найдено
`StrRScan(Str,Chr)` `AnsiStrRScan(Str, Chr)`	Возвращает указатель на последнее вхождение символа `Chr` в PChar-строку `Str`	`Str` — PChar-строка; `Chr` — символ	PChar-строка, начиная с последнего вхождения символа `Chr` в строку `Str`; `nil` — если вхождение не найдено
`StrScan(Str,Chr)` `AnsiStrScan(Str,Chr)`	Возвращает указатель на первое вхождение символа `Chr` в PChar-строку `Str`	`Str` — PChar-строка; `Chr` — символ	PChar-строка, начиная с первого вхождения символа `Chr` в строку `Str`; `nil` — если вхождение не найдено
`StrUpper(Source)` `AnsiStrUpper(Source)`	Преобразование символов PChar-строки к верхнему регистру (все прописные)	PChar-строка `Source`	Преобразованная PChar-строка

Примеры использования функций:

1. Рассмотрим фрагменты программы:

```
var
  VPStr1, VPStr2: PChar;   // Объявление переменных типа PChar
  S: string;
```

Следующий условный оператор проверяет вхождение PChar-строки VPStr1 в PChar-строку VPStr2 и заносит значение в строку S:

```
  if StrPos(VPStr1, VPStr2) <> nil then
      S:='Подстрока найдена'
  else S:= 'Подстрока не найдена';
```

Если перед выполнением оператора присвоены значения VPStr1= 'Object Pascal', а VPStr2='Pascal', то в строку S занесется значение 'Подстрока найдена'.

2. Пример извлечения имени файла из полного пути.

```
var    // Объявления переменных
  FileName, P: PChar;
  S: string;
```

Фрагмент программы:

```
P := StrRScan(FileName, '\'); // Поиск последнего символа '\'
        // После выполнения оператора в переменной P
        // либо имя файла, либо nil
if P = nil then
begin
 P := StrRScan(FileName, ':'); // Поиск последнего символа ':'
             // После выполнения оператора в переменной P
             // либо имя файла, либо nil
 if P = nil then P := FileName;
end;
S := string(P);     // Занесение в S имени файла
```

Запись

Тип запись содержит объединенный общим именем набор компонентов или полей, которые могут быть различных типов. Объявление типа *запись* задает для каждого поля идентификатор, который именует поле, и тип данных поля.

Синтаксис:

```
<Имя типа> = record
<Список имен полей> : < Тип >;
...
<Список имен полей> : < Тип >;
end;
```

В соответствии с приведенным синтаксическим описанием поля в объявлении записи отделяются друг от друга точкой с запятой, а при наличии нескольких полей с одним и тем же типом данных идентификаторы этих полей могут быть описаны в одной строке и перечислены при этом через запятую. Количество полей записи не ограничено.

После объявления записи можно задать переменные или константы этого типа, например:

```
type
TDateRec = record
    Year: Integer;
    Month: 1 .. 12;
    Day:  1 .. 31;
    end;
TPerson = record
    FirstName, SecondName:string[20];
    Sex: (Man, Woman);
    Age: Integer
    end;
var
    Person:TPerson;
    DateRec:TDateRec;
```

Запись можно объявлять и непосредственно в объявлении переменной (в разделе `var`), не прибегая к объявлению типа:

```
<Имя переменной> = record
<Список имен полей> : < Тип >;
...
<Список имен полей> : < Тип >;
end;
```

Доступ к полям записи осуществляется путем указания имени переменной (или константы) и имени поля, разделенными точкой. Например, для переменных `Person` и `DateRec`, объяв-

ленных выше, операторы присваивания значений полям записываются следующим образом:

```
Person.FirstName:= 'Victor';
Person.Sex := Man;
Person.Age := 20;
DateRec.Year := 2007;
```

Объявление записи может иметь так называемую *вариантную* часть, воспринимаемую программой по-разному, в зависимости от текущего назначения. На самом деле, вариантная часть представляет собой фрагмент памяти, который может рассматриваться в разных случаях как разный набор полей. Вариантная часть в записи может быть только одна и в описании располагается в конце записи.

Синтаксис записи с вариантной частью следующий:

```
<Имя типа> = record
    <Список имен полей фиксированной части> : < Тип >
    ...
    <Список имен полей фиксированной части > : < Тип >
    case <Имя переменной выбора варианта> : <Перечисляемый тип > of
    <Значение переменной выбора варианта>:(<Список полей>);
    ...
    <Значение переменной выбора варианта>:(<Список полей>);
    end
```

В записи с вариантной частью фиксированная составляющая может отсутствовать.

Переменная выбора варианта — это всегда переменная некоторого перечисляемого типа. Вариантная часть записи воспринимается как набор полей, соответствующий значению переменной выбора, поэтому доступ к информации может быть осуществлен более чем одним способом. При этом доступ к вариантным и фиксированным полям один и тот же.

Рассмотрим несколько примеров записей с вариантной частью:

```
Type
   TProfesion = (Student, Engeneer);
   TPerson = record
     FirstName, LastName: string[20];
     BirthDay: DateRec;
```

```
    case Profesion: TProfesion of
        Student: (Course: Integer;
                  (Faculty : string[20]);
        Engeneer: (Speciality: string[30])
  end;

  TFigure = (Rectangle, Triangle, Circle);
  TColor = (Red, Green, Yellow);
  TPolygon = record
      X, Y: Real;
      Color: TColor;
      case Kind: TFigure of
      Rectangle: (RHeight, RWidth: Real);
      Triangle: (TSizel, TSize2, Angle: Real);
      Circle: (Radius: Real);
  end;
var
  Student_Person, Engeneer_Person: TPerson;
  MyRect, MyTriangle: TPolygon;
```

При использовании записи нужный вариант однозначно определяется именем поля:

```
Student_Person.Course := 5;
Engeneer_Person.Speciality := 'Programmer';
MyTriangle.Angle := 1.0;
```

При объявлении типа записи с вариантной частью допускается не вводить специальную переменную выбора, а использовать стандартные перечисляемые типы и их возможные значения, например:

```
Var
  TConvert = record
      case Integer of
      1:(CPoint : Longint);
      2:(Carray : array [1..4] of Byte)
    end;
```

Константы с типом запись. Объявление константы с типом запись содержит идентификатор и значение каждого поля, заключенные в скобки и разделенные точками с запятой.

Рассмотрим несколько примеров констант-записей:

```
type
    TPoint = record
          X, Y : Real;
      end;
```

```
    TVector = array [0..1] of TPoint;
    TMonth = (Jan,Feb,Mar,Apr,May,Jun,Jly,Aug,Sep,Oct,Nov,Dec);
    TDateRec = record
          Day : 1..31;
          Month : TMonth;
          Year : 1950..2050;
      end;

const
  Origin : TPoint = (X : 0.0; Y : 0.0);
  Line   : TVector= ((X : -3.1; Y : 1.5), (X : 5.8; Y : 3.0));
  SomDay : TDateRec  = (Day: 2; Month: Dec; Year: 2002);
```

Поля должны указываться в том же порядке, как они следуют в объявлении типа. Если запись содержит вариантную часть, в константе можно указывать только поля одного выбранного варианта.

Файл

Тип данных файл описывает файл как линейную последовательность однотипных компонентов, размещенных на внешнем запоминающем устройстве, которые могут иметь любой тип данных за исключением типа файл или структурированного типа, содержащего компоненту с типом файл. В отличие от массива длина файла, т. е. количество компонентов, не задается, а место элемента не определяется индексом.

Синтаксис объявления типа данных файл следующий:

<Идентификатор типа> = **file of** <Тип компонента>;

Если слово `of` и тип компонента опущены, то объявляется нетипизированный файл. Нетипизированные файлы используются для доступа к любому файлу на внешнем устройстве независимо от его внутреннего формата.

Стандартный файловый тип `Text` определяет файл, содержащий символы, объединенные в строки. Следует иметь в виду, что тип `Text` не идентичен типу `file of char`.

Приведем примеры объявлений файловых типов:

```
type
   FileNumber = file of Longint;    // Файл длинных целых чисел
   FileDigit = file of '0'.. '9';   // Файл символов-цифр
   TPoint = record
         X,Y:real
       end;
   FilePoint = file of TPoint;      // Файл записей TPoint
```

Объявив файловый тип, можно определить переменные файлового типа:

```
var
  File1: FileNumber;
  File2: FilePoint;
  File3: text;     // Файл стандартного типа text
  File4: file;     // Переменная нетипизированного файла
```

Переменные файлового типа имеют специфическое применение. Над ними нельзя выполнять никаких операций (например, присваивать значение, сравнивать), они используются лишь в качестве параметров специальных процедур работы с файлами.

Множество

Диапазон значений типа множество представляет собой множество всевозможных сочетаний объектов заданного порядкового типа. В этом случае заданный порядковый тип называется базовым типом.

Каждое возможное значение типа множество является подмножеством возможных значений базового типа.

Переменная типа множество может принимать как все значения множества, так и ни одного.

Синтаксис объявления типа множества:

```
<Имя типа> = set of <Порядковый тип>;
```

Базовый тип не должен иметь более 256 возможных значений, и порядковые значения верхней и нижней границы базового типа не должны превышать диапазона от 0 до 255. В силу этого базовыми типами множества не могут быть типы `Smallint`, `Integer`, `LongInt`, `Word`, `LongWord`, `Cardinal`.

Любой множественный тип может принимать значение [ ], которое называется пустым множеством.

Примеры множеств:

```
type
  Number = set of '0'..'9';
  Digit = set of 0..9;
  Oper = set of (Plus, Minus, Multiply, Divide);
```

Тип множество можно ввести и непосредственно при объявлении переменных:

```
var
   Digit :set of 0..9;
```

Значение переменной типа множество присваивается путем перечисления в квадратных скобках через запятую некоторых значений (или диапазонов значений) базового типа, например:

```
Digit := [1,3,5,7,9];
Digit := [0..5, 7, 8];
```

Константы типа множества. Объявление константы типа множество может содержать несколько элементов, заключенных в квадратные скобки и разделенных запятыми. Каждый элемент такой константы представляет собой константу или диапазон, состоящий из двух констант, разделенных двумя точками, например:

```
type
   Digits  = set of 0..9;
   Letters  = set of 'A'..'Z';
const
   EvenDigits : Digits = [0, 2, 4, 6, 8];
   Vowels     : Letters= ['A', 'E', 'I', 'O', 'U', 'Y'];
   HexDigits  : set of '0'..'z' = ['0'..'9', 'A'..'F',
                                   'a'..'f'];
```

Класс

Object Pascal — объектно-ориентированное расширение языка Pascal, основанное на введении понятия *класса* и *объекта* (см. главу 1, п. 1.8). Класс — тип данных, определяемый пользователем. Класс должен быть объявлен с областью видимости «программа» или «модуль». Чтобы определить в Object Pascal новый класс данных, используется следующая синтаксическая конструкция:

```
type <Имя класса> = class (<Имя предка>)
<Список членов класса>
end;
```

где <Имя класса> — любой идентификатор; необязательное <Имя предка> — идентификатор класса, свойства и методы которого

наследует объявляемый класс; <Список членов класса> — перечень полей данных, свойств и методов класса. Если <Имя предка> при объявлении класса не указывается, это означает, что класс является непосредственным наследником наиболее общего из предопределенных в Delphi классов — класса TObject.

<Список членов класса> содержит следующие разделы объявления членов класса:

```
public
    <методы, свойства, события>
published
    <свойства>
protected
    <поля, методы, свойства, события>
private
    <поля, методы, свойства, события>
```

Основное назначение разделов объявления — указание доступа к объявленным членам класса (табл. 4.8).

Таблица 4.8. **Назначение разделов объявления класса**

Раздел	Описание	Назначение
public	Открытый интерфейс класса	Содержит объявления, которые доступны для внешнего использования
published	Публикуемый	Содержит открытые свойства, т. е. свойства, которые появляются в процессе проектирования на странице свойств инспектора объектов и могут быть установлены в процессе проектирования
protected	Защищенный	Содержит объявления, доступные только для потомков объявляемого класса
private	Закрытый	Содержит объявления членов класса, доступных только внутри этого класса

Объявление полей по синтаксису совпадает с объявлением переменных или полей записи:

<Имя поля >: <Тип поля>

Поля данных (в соответствии с принципом инкапсуляции) всегда должны быть защищены от несанкционированного доступа, поэтому объявляются в разделах `private` и `protected`.

Доступ к полям, как правило, должен осуществляться через свойства, объявляющие методы чтения и записи полей. Синтаксис объявления свойств следующий:

```
property <Имя свойства>: <Тип поля>
    read <Имя поля или метода чтения> write <Имя поля
                                    или метода записи>
```

Если после `read` или `write` объявлено имя поля, это означает прямое чтение или запись данных. Объявление метода указывает на доступ к полю посредством специфицированных функции для чтения и процедуры для записи. При этом функция не имеет параметров и возвращает значение того типа, который указан для свойства, а процедура имеет один параметр того же типа, например:

```
type
  ExampleClass = class (TObject)
    ptivate
      FTag: Longint;
    protected
      procedure SetTag(ValTag: Longint); // Процедура записи
    published
      property Tag: Longint read FTag write SetTag;
  end;
```

При объявлении нового класса как класса-наследника становятся доступны все методы класса-родителя. Помимо этого можно объявлять новые методы или переопределять или перегружать наследуемые.

Методы могут быть статическими, виртуальными, динамическими и абстрактными.

Статическими методы являются по умолчанию. Такие методы вызываются как обычные подпрограммы. Переопределение статического метода в классе-наследнике полностью перекрывает метод класса-родителя для объектов класса-наследника (при этом тип метода может измениться).

Виртуальные и *динамические* методы реализуют правило полиморфизма объектного программирования. Они не отменяют методы с одинаковыми именами в классах-наследниках: если в

классах-наследниках методы перекрыты, вызываться во время выполнения из всех методов с одинаковыми именами будет всегда тот, который соответствует классу объекта, указанного при непосредственном вызове. Все перекрывающие друг друга виртуальные и динамические методы с одинаковым именем должны быть одного типа. Различие между виртуальными и динамическими методами — в особенностях поиска адреса при вызове. Динамические метоты экономичнее с точки зрения ресурсов памяти, а виртуальные — с точки зрения быстродействия.

Абстрактными называются методы, которые объявлены в классе, но не содержат никаких действий, никогда не вызываются и обязательно должны быть переопределены в потомках класса. Абстрактными могут быть только виртуальные и динамические методы.

Синтаксис объявления метода в классе следующий:

```
<Тип подпрограммы>; [reintroduce;] [overload;]
                    [virtual|dynamic|override; [abstract;]]
```

Назначение ключевых слов приведено в табл. 4.9.

Таблица 4.9. **Назначение ключевых слов в объявлении метода**

Ключевое слово	Назначение
`virtual`	Объявление виртуального метода
`dynamic`	Объявление динамического метода

`override`	Объявление перегружаемого в классе-наследнике виртуального или динамического метода (не может использоваться для статических методов!), например:
	`TFirstClass = class`
	`Field1: Integer;`
	`procedure VirtMethod1; virtual;`
	`procedure DynMethod1; dynamic;`
	`end;`
	`TSecondClass = class (TFirstclass)`
	`Field1: Integer;`
	`procedure VirtMethod1; override;`
	`procedure DynMethod1; override;`
	`end;`

`abstract`	Объявление абстрактного метода, например:
	`TFirstClass = class`
	`Field1: Integer;`
	`procedure VirtMethod1; virtual; abstract;`

Окончание табл. 4.9

Ключевое слово	Назначение

overload	Объявление перегружаемого метода (для одинаковых действий с разнотипными данными). Применяется для методов любого типа, например:
	`TFirstClass = class`
	`FIntData: Integer;`
	`procedure SetData(FValue:Integer); overload;`
	`end;`
	`TSecondClass = class (TFirstClass)`
	`FStringData: Double;`
	`procedure SetData(FValue:string); overload;`
	`end;`
	`...`
	`var Obj2: TSecondClass;`
	`...`
	`Obj2.SetData('Текстовая строка')`
	`// вызывается метод из TSecondClass`
	`Obj2.SetData(2) // вызывается метод из TFirstClass`

reintroduce	Объявление перегружаемого виртуального метода
	`TFirstClass = class`
	`FIntData: integer;`
	`procedure SetData(FValue:Integer); overload;`
	`virtual;`
	`end;`
	`TSecondClass = class (TFirstClass)`
	`FStringData: Double;`
	`procedure SetData(FValue:string); reintroduce;`
	`overload;`
	`end;`

При реализации метода, переопределенного любым способом в классе-наследнике, можно вызвать метод класса-родителя. Для этого перед именем метода при вызове добавляется ключевое слово `inherited`, например:

```
inherited Create(...);
```

вызывает метод `Create` родителя.

Для класса объявляются специальные методы, создающие и уничтожающие объект.

Методы, создающие и инициализирующие объект, называются *конструкторы*. Объявление конструктора выглядит как объявление процедуры с ключевым словом `constructor`:

```
constructor <Имя процедуры> [<Параметры>];
```

Методы, уничтожающие объект и освобождающие занимаемую им память, называются *деструкторы*. Объявление деструктора выглядит как объявление процедуры с ключевым словом `destructor`:

```
destructor <Имя процедуры>;
```

В качестве имени деструктора обычно используют имя `Destroy`, например:

```
destructor Destroy;
destructor Destroy; override;
```

4.1.4. Тип данных `Variant`

В переменных типа `variant` могут храниться данные любых типов, кроме `Int64`, указателей, статических массивов, записей, множеств, классов, файлов. Такие переменные могут также хранить объекты COM и CORBA, обеспечивая доступ к их методам и свойствам.

Переменная типа `variant` при объявлении не имеет типа. Тип задается оператором присваивания переменной некоторого значения и становится равным типу этого значения, например:

```
var V : Variant;
V := 100;      // Тип V становится равным integer
V := 'Переменная variant' // Тип V становится равным string
V := true;     // Тип V становится равным boolean
```

Размер переменной типа `variant` — 2 байта, в которых содержатся код типа и значение переменной (или указатель на значение). В момент создания переменная инициализируется специальным значением `Unassigned`. Узнать текущий тип переменной `variant` можно с помощью функции `VarType`, которая возвращает хранящееся в переменной значение типа. Предопределенная константа `VarTypeMask` позволяет преобразовать (с помощью операции умножения `and`) значение, возвращаемое функцией `VarType`, в константное значение типа. Например, выражение:

```
VarType(V) and VarTypeMask = varInteger
```

имеет значение `true`, если в переменной `V` в данный момент хранится значение типа `Integer`. Некоторые константные значения типов приведены в табл. 4.10.

Таблица 4.10. **Некоторые константные значения типов**

Константа	Числовое значение	Тип
varEmpty	$0000	Состояние Unassigned
varNull	$0001	Значение переменной равно null
varIntegr	$0003	Тип Integer
varSingle	$0004	Тип Single
varDouble	$0005	Тип Double
varWord	$0012	Тип Word
varString	$0100	Тип string
varArray	$2000	Ссылка на динамический массив variant
varTypeMask	$0FFF	Битовая маска для извлечения кода типа

Для размещения динамических массивов типа variant используются функции VarArrayCreate и VarArrayOf.

Функция VarArrayCreate объявлена следующим образом:

```
function VarArrayCreate(const Bounds: array of Integer;
                        VarType:Integer):Variant;
```

Параметр Bounds представляет массив, содержащий четное количество целых значений, каждая пара которых последовательно определяет нижнее и верхнее значения индексов массива; параметр VarType определяет тип элементов массива, например:

```
var Ar, vAr: Variant;
...
Ar := VarArrayCreate([0,99], varWord);
                      // Функция размещает переменную Ar
                      // типа массив из 100 элементов Word
vAr := VarArrayCreate([0,2], varVariant);
                      // Функция размещает переменную vAr
                      // типа массив из 3 элементов variant
// Заполнение элементов массива значениями различных типов
vAr[0] := 1;
vAr[1] := 'Редактировать';
vAr[2] ;= true;
```

Функция VarArrayOf объявлена следующим образом:

```
function VarArrayOf(const Values:array of Variant):Variant;
```

Она возвращает одномерный массив элементов, задаваемый параметром Values, например:

```
vAr[0] := VarArrayOf([2, 'Копировать', false]);
```

4.1.5. Тип данных указатель

Тип данных указатель задает множество значений, которые соответствуют адресам переменных определенного типа, называемого базовым типом.

Синтаксис объявления типа:

```
<Имя типа-указателя> = ^<Имя базового типа>;
```

Например:

```
type
   TCoord = record
     X, Y: Real
   end;
   PTCoord = ^TCoord;
var
   P1, P2, P3 : PTCoord;
   Point : TCoord;
   P : Pointer;
```

При объявлении переменной типа PTCoord резервируется память под физический адрес переменной типа TCoord (но не под сам тип TCoord).

Присвоить значение переменной типа указатель можно двумя путями:

во-первых, разместить в памяти новую (динамическую) переменную с помощью стандартной процедуры New с параметром типа указатель, в который после выполнения процедуры будет занесено значение адреса размещения;

во-вторых, присвоить значение адреса размещения объявленной переменной базового типа (т. е. статической переменной), используя знак операции «@».

Процедура `New` отводит новую область в динамической памяти для переменной и сохраняет адрес этой области в переменной типа указатель.

Знак операции «@» устанавливает переменную типа указатель на область памяти, содержащую объявленную переменную.

Зарезервированное слово `nil` обозначает константу с пустым значением указателя.

Встроенный тип `Pointer` обозначает нетипизированный указатель, т. е. указатель, который не указывает ни на какой определенный тип. Как и значение, обозначаемое словом `nil`, значения типа `Pointer` совместимы со всеми другими типами указателей.

Для приведенных выше объявлений справедливы следующие операторы:

```
New(P1);
P2 := @Point;
New(P);
P3 := P;
```

В языке определен ряд стандартных процедур и функций для переменных типа указатель. Перечень их представлен в табл. 4.11.

Таблица 4.11. **Стандартные процедуры и функции для переменных типа указатель**

Название	Статус	Назначение	Параметры	Результат
`Addr(X)`	F	Возвращает адрес статической переменной	`X` — параметр любого типа (включая процедурный)	Указатель типа `Pointer`
`Ptr(Address)`	F	Преобразует значение целого типа в указатель	`Address` — параметр типа Integer	Указатель типа `Pointer`
`New(P)`	P	Размещает в памяти динамическую переменную типа, базового для параметра P. Размер выделяемой памяти совпадает с размером базового типа	`P` — параметр типа указатель	В параметр `P` заносится значение адреса переменной базового типа
`Dispose(P)`	P	Освобождение памяти, на которую указывает параметр. Размер освобождаемой памяти совпадает с размером базового типа	`P` — параметр типа указатель	После выполнения процедуры значение `P` не определено

Окончание табл. 4.11

Название	Статус	Назначение	Параметры	Результат
`GetMem(P,Size)`	P	Выделяет память заданного размера под переменную	`P` — параметр типа указатель (включая тип `Pointer`); `Size` — параметр типа `Integer`	В параметр `P` заносится значение адреса переменной
`FreeMem(P[,Size])`	P	Освобождает память заданного размера, занятую переменной, на которую указывает параметр. Если размер не указан, освобождается память, предварительно выделенная с помощью процедуры `GetMem`	`P` — параметр типа указатель (включая тип `Pointer`); `Size` — параметр типа `Integer`	После выполнения процедуры значение `P` не определено

Указатели, определяющие адреса переменных разного типа, сами являются переменными разного типа, и для них не допустимо использование оператора присваивания. Исключение составляет тип `Pointer`, который совместим по присваиванию с любым другим типом указатель. В Object Pascal имеется ряд предопределенных типов указателей (табл. 4.12).

Таблица 4.12. **Предопределенные типы указателей**

Тип указателя	Тип переменной, доступной по указателю
`PAnsiString`, `PString`	`AnsiString`
`PByteArray`	`ByteArray` (объявлен в модуле `SysUtils`). Используется для доступа к динамически размещаемым массивам байтов
`PCurrency`	`Currency`
`PExtended`	`Extended`
`POleVariant`	`OleVariant`
`PShortString`	`ShortString`
`PTextBuf`	`TextBuf` (объявлен в модуле `SysUtils`). Тип буфера в файловой записи `TTextRec`

Окончание табл. 4.12

Тип указателя	Тип переменной, доступной по указателю
PVarRec	TVarRec (объявлен в модуле System)
PVariant	Variant
PWideStr	WideStr
PWordArray	TWordArray (объявлен в модуле SysUtils). Используется для доступа к динамически размещаемым массивам с элементами типа Word

К указателям одного типа можно применять операции сравнения = и <>.

Чтобы получить непосредственно значение переменной, адрес которой содержится в указателе, необходимо после имени указателя поставить знак «^», например:

```
Point := P1^;
P2^ := Point;
```

Константы с типом указатель

Объявление константы типа указатель обычно содержит знак операции «@» и идентификатор константного значения некоторого типа, например:

```
type
   TDirection = (Left, Right, Up, Down);
   TNodePtr = ^Node;
   TNode = record
           Value : TDirection;
           Next : TNodePtr;
         end;
const
   N1: TNode = (Value: Down;Next: nil);
   N2: TNode = (Value: Up;  Next: @N1);
   N3: TNode = (Value: Right; Next: @N2);
   N4: TNode = (Value: Left;Next: @N3);
   DirectionTable: TNodePtr = @N4;
```

В приведенном примере необходимо обратить внимание на то, что синтаксис языка разрешает объявлять тип-указатель перед объявлением базового типа, например, в случае, когда речь идет о формировании последовательностей структур данных типа запись со ссылками друг на друга.

Работа с динамическими структурами данных

Рассмотрим примеры создания динамических структур данных, т. е. структур данных с переменным числом элементов. Чаще всего при решении практических задач используют связанные линейные (списки) или иерархические структуры (деревья). Каждый элемент такой структуры состоит всегда из двух частей:

- собственно информационная часть, содержащая данные, подлежащие обработке;
- ссылочная часть, содержащая набор ссылок на соседние элементы структуры (количество и содержимое ссылок диктуется разновидностью структуры).

Наилучшим для хранения такого рода элементов представляется тип данных запись, в которой одно или несколько полей отведены под указатели (см. описание записи `TNode`). В этом случае объявление типа указатель должно предшествовать объявлению типа запись, как показано в предыдущем примере, и синтаксис языка это разрешает.

При работе с динамическими структурами данных необходимо программировать выполнение следующих основных операций:

- добавление элемента в структуру;
- исключение элемента из структуры;
- поиск определенного элемента структуры по заданному признаку.

Рассмотрим задачу формирования линейной структуры, называемой однонаправленным списком (см. п. 1.4 гл. 1). Это структура, каждый элемент которой имеет ссылку на следующий за ним. У последнего элемента такая ссылка равна пустому значению (`nil`).

Пусть элемент структуры в информационной части содержит строку. Тогда тип такого элемента можно описать следующим образом:

```
type
   TNodePtr = ^TNode;
   TNode = record
       Value : string;
       Next : TNodePtr;
         end;
```

Для обеспечения работы со списком введем следующие переменные:

```
var
  NodeB, NodeE, NodeCur: TNodePtr;
  S: string;
```

где NodeB — указатель на первый элемент списка, NodeE — указатель на последний элемент списка, NodeCur — указатель на текущий элемент списка.

Перед формированием списка необходимо обнулить указатели на начало и конец списка, чтобы показать, что в списке нет элементов:

```
NodeB := nil;
NodeE := nil;
```

Теперь необходимо разработать процедуры, выполняющие основные операции работы со списком.

Начнем с процедуры внесения элемента в список. Чтобы внести новый элемент в однонаправленный список со ссылкой на следующий, необходимо знать адрес того элемента, после которого необходимо добавить новый. Значит, одним из параметров процедуры должен быть указатель типа TNodePtr. В качестве другого параметра укажем значение информационной части (параметр типа string).

Объявление процедуры имеет вид (проверка Node на значение nil вынесена за рамки процедуры):

```
procedure NewNode(Node: TNodePtr; Inf: string);
```

Процедура должна обеспечить выполнение следующих действий:

1. Выделение фрагмента памяти под новый элемент:

```
New(N);  // N - локальная переменная типа TNodePtr
```

2. Занесение в новый элемент информации:

```
N^.Value := Inf;
```

3. Перенос в поле ссылки на следующий нового элемента значения из элемента Node:

```
N^.Next := Node;
```

4. Занесение в поле ссылки на следующий Node адреса размещения нового элемента:

```
Node^.Next := N;
```

Общее описание процедуры:

```
procedure NewNode(Node: TNodePtr; Inf: string);
var
     N: TNodePtr;
begin
   New(N);
   N^.Value := Inf;
   N^.Next := Node;
   Node^.Next := N;
end;
```

По аналогии с процедурой внесения элемента опишем процедуру удаления элемента, размещенного после элемента Node (проверка Node на значение nil вынесена за рамки процедуры):

```
procedure DeleteNode(Node: TNodePtr);
var
     N: TNodePtr;
begin
   N := Node^.Next;
      // Занесение в переменную N адреса удаляемого элемента
   Node^.Next := N^.Next;
      // Обеспечение обхода по ссылкам удаляемого элемента
   Dispose(N);    // Освобождение памяти
end;
```

Последняя операция — поиск элемента по заданному значению — может быть реализована с помощью функции, выполняющей такую последовательность действий:

```
function SearchNode(inf: string): TNodePtr;
var
     N: TNodePtr;
begin
  N := NodeB; // Занесение в переменную N адреса начала списка
  while N <> nil do // Организация цикла по элементам списка
     if (N^.Value = inf) then Break
                  // Break — досрочный выход из цикла
     else N := N^.Next;
                  // Обеспечение перехода на следующий элемент
  SearchNode := N;
     // Результат функции — nil или адрес найденного элемента
end;
```

Приступим к формированию списка. Для размещения первого элемента используем процедуру New:

```
New(NodeB);
NodeB^.Value := S; // В переменную S предварительно занесено
                   // значение информационной составляющей элемента
NodeB^.Next := nil;
NodeE := NodeB;
```

Добавить новый элемент в конец списка теперь можно с помощью процедуры:

```
NewNode(NodeE, S);
```

Пусть теперь необходимо найти элемент, в поле Value которого находится значение 'Волга', и вставить после него в список элемент со значением 'Урал' в поле Value. Эта задача решается следующей последовательностью операторов:

```
NodeCur := SearchNode('Волга');
If NodeCur <> nil then NewNode(NodeCur, 'Урал');
```

Удалим теперь из списка элемент, следующий за элементом со значением 'Ока' в поле Value, если у следующего за ним элемента значение в поле Value то же самое:

```
NodeCur := SearchNode('Ока');
if NodeCur <> nil then
       // Если элемент со значением 'Ока' в поле Value найден
   if NodeCur^.Next <> nil then // если есть следующий элемент
     if NodeCur^.Next^.Value = 'Ока'
     // Если в поле Value следующего элемента - значение 'Ока'
     then DeleteNode(NodeCur);
```

Первый элемент из списка удаляется путем переназначения первого элемента:

```
NodeCur := NodeB;
NodeB := NodeB^.Next;
Dispose(NodeCur);
```

Для удаления последнего необходимо сначала найти предпоследний, т. е. тот, у которого в поле Next стоит значение NodeE:

```
NodeCur := NodeB;
While NodeCur^.Next <> NodeE do
      NodeCur := NodeCur^.Next;
```

Затем необходимо переопределить указатель на последний элемент и удалить последний элемент:

```
NodeE := NodeCur;
DeleteNode(NodeCur);
```

4.1.6. Выражения и операции

Допустимое множество операций языка включает следующие группы операций:

- арифметические;
- операции отношения;
- логические;
- операции с битами информации;
- операции со строками;
- операции с указателями;
- операции с множествами;
- операции с классами.

Каждой группе операций соответствуют определенные типы переменных и констант.

Порядок, в котором выполняются операции, соответствует общему приоритету операций (см. гл. 1, п. 1.2).

Арифметические операции могут применяться только к операндам целых и вещественных типов. В качестве операндов арифметических операций могут выступать стандартные арифметические (или тригонометрические) функции, т. е. функции с результатом целого или вещественного типа. В табл. 1.1 приведены арифметические операции языка, а в табл. 1.2 — характеристики некоторых арифметических и тригонометрических функций.

В результате выполнения *операций отношения* получается значение логического типа — `true` или `false`. Все операции отношения — бинарные. Перечень допустимых операций приведен в табл. 1.3.

Логические операции применяются к операндам логического типа. Правила выполнения логических операций приведены в табл. 1.4.

Операции с битами информации работают с двоичным представлением целых чисел. Перечень допустимых операций с битами информации и правила их выполнения приведены в табл. 1.5, 1.6.

Допустимые операции для других типов данных (операции со *строками, указателями, множествами* и *классами*) представлены в табл. 4.13.

Таблица 4.13. **Допустимые операции для некоторых типов данных**

Операция	Назначение	Тип операндов	Тип результата	Действие
Операции со строками				
+	Конкатенация	Строковый, символьный	Строковый	`S+T` — за символами первой строки в результате будут следовать символы второй строки
Операции с указателями				
+	Сложение	Указатель, целое	Указатель	`P+I` — увеличение адреса, на который указывает `P` (указатель), на `I` (целое) байт
–	Вычитание	Указатель, целое	Указатель, целое	`P-I` — уменьшение адреса, на который указывает `P` (указатель), на `I` (целое) байт; `P-Q` — разность между указателями `P` и `Q` в байтах
^	Разименование	Указатель	Базовый тип	`P^` — обеспечивает доступ к переменной, на которую указывает `P`
=	Равенство	Указатель	Логический	`P=Q` дает значение `true` только в том случае, когда `P` и `Q` указывают на один адрес
<>	Неравенство	Указатель	Логический	`P<>Q` дает значение `true` только в том случае, когда `P` и `Q` не указывают на один адрес
@	Возвращает адрес операнда	Любой	Указатель `Pointer`	`@MyVariable` — возвращает адрес размещения переменной `MyVariable`
Операции с множествами				
+	Объединение	Множества	Множество	Значение `C` принадлежит `A+B` только в том случае, если `C` принадлежит `A` или `B`
–	Разность	Множества	Множество	Значение `C` принадлежит `A-B` только в том случае, если `C` принадлежит `A`, но не принадлежит `B`
*	Пересечение	Множества	Множество	Значение `C` принадлежит `A*B` только в том случае, если `C` принадлежит и множеству `A`, и множеству `B`

Окончание табл. 4.13

Операция	Назначение	Тип операндов	Тип результата	Действие
= ~	Эквивалентность	Множества	Логический	Выражение `A = B` истинно только тогда, когда `A` и `B` содержат одни и те же элементы
<>	Не эквивалентность	Множества	Логический	Выражение `A <> B` истинно только тогда, когда хотя бы один элемент множества `A` не содержится во множестве `B` или наоборот
<=	Подмножество	Множества	Логический	Выражение `A <= B` истинно, если каждый элемент множества `A` является также и элементом множества `B`
>=	Включающее множество	Множества	Логический	Выражение `A >= B` истинно, когда каждый элемент множества `B` является также и элементом множества `A`
in	Является элементом	Множества	Логический	Проверка на принадлежность множеству. Выражение `x in A` истинно, когда значение элемента порядкового типа является элементом типа множества, в противном случае выражение ложно
Операции с классами				
=	Эквивалентность	Класс	Логический	`C1=C2` — возвращает `true`, если классы эквивалентны
<>	Не эквивалентность	Класс	Логический	`C1<>C2` возвращает `true`, если классы не эквивалентны
as	Рассматриваемый, как	Первый — объект, второй — класс	Объект	`A as C` — возвращает объект `A`, рассматриваемый как объект класса `C`
in	Относящийся к	Первый — объект, второй — класс	Логический	`A in C` — возвращает `true`, если объект `A` относится к классу `C`

Следует отметить, что результат операции сложения символов или строк (конкатенации) совместим с любым строковым типом (но не с символьным). Если оба операнда имеют тип ко-

роткой строки или символьный и длина результата превышает 255, результат усекается до первых 255 символов.

Результаты операций над множествами подчиняются правилам алгебры множеств. Операндами в операциях являются множества с совместимыми типами.

4.1.7. Операторы языка

Оператор присваивания

С помощью этого оператора переменной, имя которой указывается в левой части оператора, присваивается значение выражения, стоящего в правой части и вычисляемого перед присваиванием.

Синтаксис:

```
<Переменная> := <Выражение>
```

В левой части оператора присваивания может стоять переменная любого типа (кроме типа `File`) или типизированная константа, но при этом типы переменной и выражения должны совпадать.

Примеры:

1. Вычисление длины окружности `L` (тип — `Real`):

```
L := 2*Pi*R
```

`R` — радиус окружности (тип — `Integer` или `Real`);

`Pi` — константа, равная π.

2. Присвоение переменной `Flag` (тип — `Boolean`) значения `true`, если переменная `X` (тип — `real`) находится в интервале (0; 1), и `false` — в противном случае:

```
Flag := (X > 0) and (X < 1)
```

Следует помнить, что применительно к переменным некоторых типов данных (объектам, строкам) оператор присваивания не копирует содержимое самой переменной, а присваивает только ссылку на место памяти, где содержимое располагается. На-

пример, для двух объектов Object1 и Object2 выполнение оператора присваивания

```
Object1 := Object2;
```

означает, что Object1 и Object2 будут указывать на один и тот же объект. Чтобы скопировать содержимое объекта, необходимо использовать метод копирования, свойственный многим классам объектов. Синтаксис объявления метода копирования следующий:

<Объект-приемник>.Assign(<Объект-источник>;

Например:

```
Object1.Assign(Object2);
```

Оператор безусловного перехода

Оператор goto позволяет изменить стандартный последовательный порядок выполнения операторов и передать управление заданному оператору, которому в этом случае должна предшествовать *метка*. Эта же метка должна быть указана при операторе goto.

Синтаксис оператора безусловного перехода:

goto <Метка>;

Метка представляет собой целое число в пределах от 0 до 9999 либо идентификатор. Все метки должны быть перечислены в разделе объявления меток, который начинается со служебного слова Label:

```
Label 99, 100, MyLabel;
```

Одной меткой помечается только один оператор. Метка от оператора отделяется двоеточием:

<Метка>:<Оператор>

При использовании оператора безусловного перехода должны соблюдаться следующие правила.

1. Метка, указанная в операторе перехода, должна находиться в том же блоке или модуле, что и сам оператор перехода. Другими словами, не допускаются переходы из процедуры или функции или внутрь нее.

2. Переход извне внутрь составного оператора (то есть переход на более глубокий уровень вложенности) может вызвать некорректную работу программы, хотя компилятор не выдает сообщения об ошибке.

Следует также иметь в виду, что слишком частое применение оператора безусловного перехода ухудшает логику программы.

Пример:

```
if Odd(I) then
         begin X := Z * X;
                  goto 1;
         end;
I := I div 2;
X := Sqr(X);
1:;                          // Пустой оператор
```

Пустой оператор

Пустой оператор не выполняет никакого действия в программе, но может иногда потребоваться для осуществления на него безусловного перехода.

Пустой оператор может отображаться в программе точкой с запятой, отделяющей пустой оператор от предшествующего, или меткой (см. пример выше).

Оператор with

Оператор `with` используется для краткого обращения к полям записи или к свойствам и методам объекта. В операторе `with` к полям одной или более конкретных переменных типа запись или объектов можно обращаться, используя только идентификаторы полей.

Оператор `with` имеет следующий синтаксис:

```
with <Илентификатор записи или объекта> do <Оператор>;
```

Приведем пример оператора `with`:

```
type
  TForm1 = class(TForm)
     Button1: TButton;
     procedure Button1Click(Sender: TObject);
   private
    { Private declarations }
```

```
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
...
with Form1 do begin
    Button1.Caption := 'Заголовок кнопки';
    Caption := 'Заголовок формы';
end;
```

Это эквивалентно следующей записи:

```
Form1.Button1.Caption := 'Заголовок кнопки';
Form1.Caption := 'Заголовок формы';
```

В операторе `with` сначала проводится проверка каждого идентификатора переменной на соответствие полю указанной записи или объекта. Если идентификатор переменной совпадает с именем поля, то переменная всегда интерпретируется как поле записи или объекта.

Пусть обьявлены следующие переменные:

```
type
   TPoint = record
       X, Y: Integer;
   end;
var
       X : TPoint;
       Y : Integer;
```

В этом случае и к `X`, и к `Y` можно обращаться, как к переменной или как к полю записи. Например, оператор:

```
with X do
   begin
       X := 1;
       Y := 100;
   end;
```

эквивалентен последовательности операторов

```
X.X := 1;
X.Y := 100;
```

так как `X` и `Y` внутри оператора `with` воспринимаются как поля записи типа `TPoint`.

Возможна еще одна форма оператора `with`:

```
with <объект1>, <объект2>,..., <объектN> do <Оператор>;
```

Такая запись соответствует множеству вложенных конструкций:

```
with <объект1> do
     with <объект2> do
     ...
          with <объектN> do
               <Оператор>;
```

Условный оператор

В Object Pascal существует две формы условного оператора: условный оператор `If` и условный оператор `Case`.

Условный оператор if. Оператор `if` имеет следующий синтаксис:

```
if <логическое выражение> then <оператор> [else <оператор>];
```

Условный оператор в неполной форме (`if` <логическое выражение> `then` <оператор>) выполняется следующим образом: сначала вычисляется логическое выражение, затем — если значение логического выражения `true` (истина) — выполняется оператор, стоящий за `then`. Если значение логического выражения `false` (ложь), условный оператор действует как пустой. В случае использования условного оператора в полной форме (`if` <логическое выражение> `then` <оператор_true> `else` <оператор_false>) оператор_true выполняется, если значение логического выражения истинно, а оператор_false — если значение логического выражения ложно.

Рассмотрим пример.

Написать последовательность действий для решения задачи: присвоить переменной y значение $\sin x$, если $x > 0$, и значение $\cos x$, если $x \le 0$.

В этом случае необходимо использовать условный оператор в полной форме:

```
if x>0 then y:=sin(x) else y:=cos(x);
```

Изменим условие задачи следующим образом: присвоить переменной y значение $\sin x$, если $x < 1$, и значение $\cos x$, если $x \ge 2$.

Переменная *y* должна поменять свое значение только в том случае, если переменная *x* принадлежит одному из интервалов: $(-\infty, 1)$ или $[2, +\infty)$. Если же *x* лежит в интервале $[1, 2)$, переменная *y* не меняет своего значения.

Такой алгоритм может быть реализован последовательностью из двух условных операторов в неполной форме:

```
if x<1 then y:= sin(x);
if x>=2 then y := cos(x);
```

Второй оператор приведенной записи решения может быть использован в качестве оператора_else в условном операторе в полной форме. Тогда решение можно записать следующим образом:

```
if x<1 then y:= sin(x) else if x>=2 then y := cos(x)
```

Условный оператор Case. Оператор предназначен для организации выбора одной из любого количества ветвей алгоритма в зависимости от значения некоторого выражения.

Синтаксис:

```
case <выражение> of
     <список значений 1>: <оператор 1>;
     <список значений 2>: <оператор 2>;
     ...
     <список значений n>: <оператор n>;
else
     <оператор>
end;
```

Оператор с номером *i* ($1 \le i \le n$) выполняется в том случае, если выражение в заголовке оператора `Case` принимает одно из значений списка значений *i*.

При использовании оператора `Case` необходимо помнить о том, что значение выражения и константы должны быть одного типа.

Рассмотрим пример.

Присвоить строке `S` значение дня недели для заданного числа `D` при условии, что в месяце 31 день и 1-е число — понедельник.

Для построения алгоритма воспользуемся операцией `mod`, позволяющей вычислить остаток от деления двух чисел, и усло-

вием, что 1-е число — понедельник. Тогда можно записать следующий оператор Case:

```
Case D mod 7 of
    1: S := 'понедельник';
    2: S := 'вторник';
    3: S := 'среда';
    4: S := 'четверг';
    5: S := 'пятница';
    6: S := 'суббота';
    0: S := 'воскресенье';
End;
```

С помощью оператора вычисляется остаток от деления D на 7 и в зависимости от полученного значения в переменную S заносится строка, соответствующая дню недели.

В предложенной записи отсутствует оператор Else. Это объясняется тем, что выражение D mod 7 может принимать только одно из указанных значений. Запись алгоритма аналогична, например, следующей записи в полной форме (с ограничением области возможных значений переменной D типа Integer):

```
if (D>=1) and (D<=31) then
      Case D mod 7 of
          1: S := 'понедельник';
          2: S := 'вторник';
          3: S := 'среда';
          4: S := 'четверг';
          5: S := 'пятница';
          6: S := 'суббота';
      else S := 'воскресенье';
      End
else ShowMessage('Ошибка!!!');
```

Оператор цикла

Оператор цикла служит для организации выполнения циклических процессов (таких, когда одни и те же действия многократно повторяются).

Обобщенный оператор цикла имеет следующий синтаксис:

<заголовок цикла> <тело цикла>

Заголовок цикла содержит сведения об условиях выполнения циклических действий, а тело цикла представляет собой оператор — последовательность самих действий.

В Object Pascal реализовано три разновидности оператора цикла — операторы `for`, `while` и `repeat`.

Оператор for. Синтаксис:

```
for <переменная цикла> := <выражение_начало> to <выражение_конец>
do <тело цикла>
```

или

```
for <переменная цикла> := <выражение_начало> downto
                                <выражение_конец>
do <тело цикла>
```

Переменная цикла должна принадлежать счетному множеству значений (т. е. должна быть целого или перечисляемого типа). Выражение_начало, выражение_конец и переменная цикла должны быть совместимы по типу.

Выполнение оператора происходит по следующему алгоритму.

1. Вычисляется значение выражения, определяющего выражение_начало (`V_Start`), и присваивается переменной цикла (`i:=V_Start`).
2. Вычисляется значение выражения, определяющего выражение_конец (`V_Finish`).
3. Значение переменной цикла (`i`) сравнивается со значением выражения, вычисленного в п. 2 (`V_Finish`). Если `i <= V_Finish` (в случае использования оператора с перечислением `to`) или `i >= V_Finish` (в случае использования оператора с перечислением `downto`) — переход к п. 4, иначе — переход к п. 6.
4. Выполняется тело цикла.
5. В случае использования оператора с перечислением `to` переменной цикла присваивается следующее большее значение (например, если `i` целого типа, то `i:=i+1`), а в случае использования оператора с перечислением `downto` переменной цикла присваивается следующее меньшее значение (например, если `i` целого типа, то `i:=i-1`). Переход к п. 3.
6. Завершение выполнения оператора.

Из представленного алгоритма видно, что если `V_Start > V_Finish` (в случае использования оператора с перечислением `to`) или `V_Start < V_Finish` (в случае использования оператора с перечислением `downto`), то тело цикла ни разу не выполнится.

Количество шагов цикла (n_Step) может быть вычислено по следующей формуле:

для случая оператора с перечислением to:

```
n_Step = V_Finish - V_Start + 1;
```

для случая оператора с перечислением downto:

```
n_Step = V_Start - V_Finish + 1.
```

Примеры:

1. Вычислить значение функции $y = N!(y = 1 \times 2 \times 3 \times ... \times N)$ для $N = 100$.

Введем переменную целого типа y и присвоим ей начальное значение 1:

```
y := 1;
```

Далее введем переменную целого типа i, которая будет возрастать от 2 до 100, и воспользуемся оператором цикла For:

```
for i := 2 to 100 do y := y*i;
```

Число шагов цикла = 100 – 2 + 1 = 99.

2. Найти максимальный делитель натурального числа *k* (за исключением самого *k*).

Алгоритм поиска максимального делителя можно построить следующим образом:

- вычислить целую часть от деления *k* пополам и присвоить переменной *i* значение целой части;
- построить цикл с перечислением downto, V_Start=i, V_Finish=1 и оператором цикла, вычисляющим остаток от деления *k* на *i* и проверяющим значение этого остатка. Как только остаток от деления будет первый раз равен 0, необходимо присвоить максимальному делителю (D) значение *i* и закончить выполнение цикла.

Введем переменные целого типа k, i и D и построим последовательность операторов, реализующую предложенный алгоритм:

```
for i := k div 2 downto 1 do
       if k mod i = 0 then
       begin  D := i;
          goto 1
       end;
1: ;
```

Оператор while. Синтаксис:

```
while <условие> do <тело цикла>
```

Условие — это логическое выражение, истинность которого проверяется в начале каждого шага цикла. Цикл будет выполняться до тех пор, пока значение логического выражения истинно.

Если на первом же шаге значение логического выражения ложно, тело цикла не выполняется ни разу.

В отличие от оператора `for` в операторе `while` максимальное число шагов цикла заранее неизвестно. Оператор предусматривает изменение значения переменных, входящих в логическое выражение, внутри тела цикла.

Рассмотрим Пример 2 предыдущего пункта. Решение этой задачи предполагает заранее неизвестное число шагов, которое понадобится для нахождения максимального делителя, поэтому использование в алгоритме оператора `while` будет более эффективно.

```
D:=0; i := k div 2;
while D=0 do
  begin if k mod i = 0 then D := i;
            i := i-1
  end;
```

Оператор `while` будет выполняться до тех пор, пока `D = 0`, т. е. пока не найден ни один делитель. Первый же найденный делитель будет максимальным, значение его занесется в переменную `D`, которая при этом станет отличной от нуля и сделает ложным логическое выражение в заголовке цикла.

Оператор repeat. Синтаксис:

```
repeat <тело цикла> until <условие>
```

Оператор `repeat` отличается от остальных двух операторов цикла тем, что проверка условия продолжения цикла стоит после тела цикла. Это обеспечивает выполнение тела цикла хотя бы один раз.

Служебные слова `repeat` и `until` играют роль операторных скобок, поэтому тело цикла, состоящее из нескольких операторов, не нужно оформлять как составной оператор (как это необходимо делать в предыдущих операторах цикла).

Тело цикла выполняется до тех пор, пока логическое выражение, формирующее условие, не станет равным истинному значению.

Рассмотрим следующий пример.

Член ряда с номером n для $n = 1, 2, 3, \ldots$ определяется выражением $\frac{1}{n^x}$. Написать последовательность операторов для вычисления суммы членов ряда от первого до члена с наименьшим номером, не превосходящего 10^{-6}.

```
S:=0;
N:=1;
SN:=1;
repeat S:=S+SN;
     Inc(N);      // Увеличение N на единицу (N:=N+1);
     SN := 1/Power(N,X);
                  // Вычисление значения очередного члена ряда
until SN<= Power(10,-6);
```

В этом цикле сначала происходит приращение суммы, а затем вычисление значения очередного члена ряда. Это обусловлено тем, что первый же член ряда, значение которого будет больше, чем 10^{-6}, не должен быть добавлен в сумму.

Стандартные процедуры Break и Continue в операторах цикла. Процедура `Break` предназначена для осуществления принудительного досрочного выхода из цикла. Например, цикл поиска максимального делителя `D` числа `k` может выглядеть следующим образом:

```
for i := k div 2 downto 1 do
     if k mod i = 0 then
     begin  D := i;
// Принудительный выход из цикла после нахождения
// первого значения D
       Break
end;
```

Процедура `Continue` принудительно вызывает начало новой итерации цикла, даже если предыдущая еще не закончена. Цикл поиска максимального нечетного делителя числа `k` может быть таким:

```
for i := k div 2 downto 1 do
   begin if i mod 2 = 0 then Continue;
```

```
            // Переход на следующую итерацию, если i четное
    if k mod i = 0 then
        begin D := i;
            Break
        end;
end;
```

Составной оператор

Составные операторы задают порядок выполнения операторов, являющихся их элементами. Они должны выполняться в том порядке, в котором записаны.

Составные операторы обрабатываются, как один оператор, что имеет решающее значение там, где синтаксис языка допускает использование только одного оператора (например, в условном операторе `if..then..else`). Операторы заключаются в ограничители (операторные скобки) `begin` и `end` и отделяются друг от друга точками с запятой.

Синтаксис:

begin <Оператор> [;...<Оператор>] **end**;

Пример составного оператора:

```
begin
    Z : = X;
    X : = 2*Y;
    Y : = Z;
end;
```

Обработка исключительных ситуаций

В Object Pascal реализован механизм обработки исключительных ситуаций, описанный в гл. 1 п. 1.2 настоящего пособия.

Обработка завершения. Синтаксис использования ключевых слов `try`, `finally` в блоке завершения следующий:

try
<Операторы защищенного блока>
finally
<Операторы блока завершения>
end;

Рассмотрим пример процедуры, выполняющей длительные вычисления. Чтобы предупредить пользователя о возможной задержке в работе программы, курсор манипулятора «мышь» обычно устанавливают в состояние «песочные часы», а после окончания вычислений возвращают в нормальное состояние:

```
procedure TForm1.BtnClick (Sender: Tobject);
var
   I, J : integer;
begin
   Screen.Cursor := crHourglass; // Установка курсора
                                 // в состояние
                                 // "песочные часы"
   J := 0;
   for I := 100000 downto 0 do   //Блок вычислений
      J := J*J + J div I;
      Screen.Cursor := crDefault;
                    //Возврат курсора в нормальное состояние
end;
```

В приведенном алгоритме есть ошибка — когда переменная I достигнет значения 0, произойдет программное прерывание, выполнение процедуры завершиться и курсор не будет возвращен в нормальное состояние. Во избежание подобных ошибок необходимо использовать блок завершений:

```
procedure Tform1.BtnClick (Sender: Tobject);
var
   I, J : Integer;
begin
   Screen.Cursor := crHourglass; //  Установка курсора
                                 // в состояние
                                 // "песочные часы"
   J := 0;
   try                           // Защищенный блок
   for I := 100000 downto 0 do
      J := J*J + J div I;
   finally                       // Блок завершения
      Screen.Cursor := crDefault;
                    //Возврат курсора в нормальное состояние
   end
end;
```

При использовании обработчика завершения курсор будет приведен в нормальное состояние независимо от того, произошла или нет исключительная ситуация, но обрабатываться исклю-

чительная ситуация при этом не будет. Рассмотрим далее синтаксис обработки исключений.

Обработка исключений. Синтаксис обработки исключений следующий:

```
try
<Операторы защищенного блока>
except
on <Тип исключительной ситуации> do <Оператор>
end;
```

Реализуем в приведенном выше алгоритме вычислений обработчик исключений:

```
procedure TForm1.BtnClick (Sender: TObject);
var
   I, J : Integer;
begin
   Screen.Cursor := crHourglass; // Установка курсора
                                 // в состояние
                                 // "песочные часы"
   J := 0;
   try                           // Защищенный блок
   for I := 100000 downto 0 do
      J := J*J + J div I;
   except        //Блок исключений
      on EDivByZero do
         ShowMessage('Исключение - деление на 0');
                                 //Обработка исключений
   end;
   Screen.Cursor := crDefault;
                  // Возврат курсора в нормальное состояние
end;
```

В этом варианте процедуры перехватывается определенная в Delphi исключительная ситуация `EDivByZero` (деление на 0). Если в процессе вычислений произойдет деление на 0, то исключение будет обработано. Если же произойдет непредусмотренная блоком исключений ситуация (например, переполнение разрядной сетки для типа данных `Integer`), то управление будет передано в блок исключений, но никаких действий произведено не будет, так как такой тип исключения не представлен в фильтре исключений. Оператор возврата курсора в нормальное состояние будет выполнен в любом случае.

4.1.8. Процедуры и функции

В языке Object Pascal существуют две разновидности подпрограмм — процедуры и функции. Каждое объявление процедуры или функции содержит обязательный заголовок, за которым следуют разделы локальных объявлений (аналогичных разделам объявлений программы) и составной оператор (блок), реализующий алгоритм подпрограммы.

Вызов процедуры на исполнение активизируется с помощью оператора процедуры. Функция активизируется при вычислении выражения, содержащего вызов функции, а возвращаемое функцией значение подставляется в это выражение.

Объявление процедур и функций

Синтаксис объявления процедур и функций в языке Object Pascal следующий:

```
procedure <Имя процедуры> (<Список формальных параметров>);
<Тело процедуры>;

function <Идентификатор>(<Список формальных параметров>):
                                        <Тип результата>;
<Тело функции>;
```

Семантика объявлений и примеры записи процедур и функций приведены в гл. 1, п. 1.5.

Переменная Result в функции

Object Pascal облегчает программирование функций путем автоматического объявления в каждой из них локальной переменной `Result`. Эта переменная имеет тот же тип, что и результат функции. Присваивание значения переменной `Result` аналогично определению значения функции. Преимущество использования этой переменной при вычислении значения функции в том, что локальная переменная `Result` может стоять в правой части оператора присваивания и не вызывает при этом рекурсивного вычисления функции (в отличие от идентификатора функции, появление которого в вычисляемом выражении означает рекурсивный вызов функции).

Например, функция вычисления суммы квадратов натурального ряда может быть записана следующим образом:

```
function SumSqr(N: Integer): Integer;
var
  i: Integer;
begin
   Result := 1;
   for i := 2 to N do
     Result := Result +i*i;
       // После выхода из цикла в переменной Result — значение
       // функции
end;
```

Заметим, что использование `Result` уменьшает количество локальных переменных, необходимых для реализации алгоритма, и количество операторов в теле функции.

Формальные и фактические параметры

Объявление процедуры или функции содержит список формальных параметров. Каждый параметр из списка формальных параметров является локальным по отношению к процедуре или функции, для которой он объявлен. Это означает, что глобальные переменные, имена которых совпадают с именами формальных параметров, становятся недоступными для использования в процедуре или функции.

Отдельные объявления параметров в списке разделяются точкой с запятой.

Параметры-значения. Формальные параметры-значения действуют как переменные, локальные по отношению к процедуре или функции. Изменения формальных параметров-значений не влияют на значения соответствующих фактических параметров.

Синтаксис объявления параметра-значения следующий:

<идентификатор> [,...<идентификатор>]:<Тип параметра>

Фактический параметр, соответствующий параметру-значению в операторе процедуры или вызове функции, должен быть выражением, а его значение не может быть файлового типа. Фактический параметр должен быть совместим по присваиванию с типом формального параметра-значения. Пример объявления функции с параметром-значением приведен в гл. 1, п. 1.5.

Параметры-переменные. Формальные параметры-переменные служат для модификации внутри подпрограммы значений соответствующих фактических параметров. Формальный параметр-переменная представляет фактическую переменную во время выполнения процедуры или функции, поэтому все изменения значения формального параметра отражаются на фактическом параметре.

Синтаксис объявления параметров-переменных следующий:

```
var <идентификатор> [,...<идентификатор>]:<Тип параметра>
```

или

```
var <идентификатор> [,...<идентификатор>]
```

Внутри подпрограммы любое упоминание формального параметра-переменной обеспечивает доступ к самому фактическому параметру. Тип фактического параметра должен быть тождественен типу формального параметра-переменной (это ограничение можно обойти через использование параметров-переменных без типа). Файловые типы могут передаваться только как параметры-переменные. Пример объявления процедуры с параметром-переменной приведен в гл. 1, п. 1.5.

Как видно из синтаксической формулы, объявление параметров-переменных может не сопровождаться указанием типа. Описание такого способа передачи параметров и пример применения см. в гл. 1, п. 1.5.

Параметры-константы. Формальные параметры-константы носят пограничный характер между двумя категориями. С одной стороны, это параметры, которые передаются в подпрограмму по наименованию, т. е. ссылкой на глобальное размещение фактического параметра, но, с другой стороны, внутри подпрограммы действует запрет на изменение значения параметра. Использовать параметры-константы удобно вместо параметров-значений, когда параметр характеризуется большим размером занимаемой памяти.

Синтаксис объявления параметров-констант следующий:

```
const <идентификатор> [,...<идентификатор>]:<Тип параметра>
```

или

```
const <идентификатор> [,...<идентификатор>]
```

Пример объявления функции с параметром-константой см. в гл. 1, п. 1.5.

Массивы открытого типа. Массив открытого типа — это массив, который можно передавать в процедуру и функцию в качестве параметра-значения или параметра-константы без указания длины. Формальным параметром в этом случае может выступать любой массив, состоящий из элементов того же типа, что и открытый массив. Введение такого параметра сделало возможным разрабатывать подпрограммы, обрабатывающие массивы однотипных элементов любой размерности.

В объявлении открытого массива как параметра подпрограммы отсутствует указание типа индекса (в отличие от объявления простого массива). Следует также иметь в виду, что нумерация элементов открытого массива начинается с 0. Максимальное значение индекса фактического массива определяется с помощью функции `High(X)`.

Функция `High(X)` и симметричная ей функция `Low(X)` служат для получения максимального и, соответственно, минимального значения величины в зависимости от ее типа. Зависимость возвращаемых функциями значений от типа параметра `X` представлена в табл. 4.14.

Таблица 4.14. **Возвращаемые значения для функций `High(X)` и `Low(X)`**

Тип параметра `X`	`High(X)`	`Low(X)`
Простой (базовый)	Максимальное значение из области определения	Минимальное значение из области определения
Массив	Максимальное значение из области определения индекса	Минимальное значение из области определения индекса
Строка типа `String`	Объявленное количество символов в строке	0 только для строк типа `Shortstring`
Открытый массив	Значение целого типа, равное количеству элементов фактического параметра-массива без единицы (так как нумерация элементов массива начинается с нуля)	0
Параметр подпрограммы типа `String`	Значение целого типа, равное количеству символов фактического параметра-строки без единицы	0

Приведем пример использования массива открытого типа.

Пусть функция вычисления суммы положительных элементов массива вещественных чисел определена следующим образом:

```
function SumPos(var X: array of Real): Real;

var
  I: Word;
  S: Real;
begin
  S := 0;
       // Нумерация элементов открытого массива — с нуля
  for I := 0 to High(X) do
     if X[I] > 0 then S := S + X[I];
  SumPos := S;
end;
```

Тогда для объявленных в программе массивов

```
var
  List1: array [0..30] of Real;
  List2: array [5..175] of Real;
  X: Word;
  S, TempStr: string;
```

Можно использовать следующую последовательность операторов:

```
      // Умножение элементов массива на константу
for X := Low(List1) to High(List1) do
       List1[X] := X * 3.4;
for X := Low(List2) to High(List2) do
       List2[X] := X * 0.125;
// Вычисление суммы положительных элементов и преобразование
// вещественного числа в строку с указанием формата:
// 8 знаков всего, 4 знака после десятичной точки
  Str(SumPos(List1):8:4, S);
       // Вывод сообщения на экран
  ShowMessage('Сумма положительных элементов List1: ' + S);
       // Стандартная процедура ShowMessage

  Str(SumPos(List2):8:4, S);
  ShowMessage('Сумма положительных элементов List2: ' + S);
```

Процедурные типы. Процедурный тип данных позволяет рассматривать обращение к процедуре или функции как перемен-

ную, имеющую значение. Такая переменная может выступать и в качестве параметра подпрограммы.

В объявлении процедурного типа присутствует список формальных параметров и (для функции) тип результата.

<Идентификатор типа> = **procedure** <Список формальных параметров>;

или

<Идентификатор типа> = **function** <Список формальных параметров >:
<Тип результата>;

Синтаксис объявления процедурного типа совпадает с синтаксисом объявления заголовка процедуры или функции, за исключением того, что имя процедуры (функции) после ключевого слова `procedure` или `function` опускается.

Примеры объявлений процедурного типа:

```
type
   TMyProc = procedure;
   TSwapProc = procedure(var X, Y: Integer);
   TStrProc = procedure(S: string);
   TWriteFileFunc = function(var F: Text): Integer;
   TMathFunc = function(X: Real): Real;
```

Имена параметров в объявлении процедурного типа чисто декоративные — они не воздействуют на смысл объявления.

Константы процедурного типа. Константа процедурного типа должна указывать идентификатор процедуры или функции, совместимый по присваиванию с типом константы.

Пример:

```
type
   TErrorProc = procedure(ErrorCode: Integer);

procedure DefaultError(ErrorCode: Integer);
begin
   ShowMessage('Error '+ inttostr(ErrorCode));
     // Стандартная процедура преобразования типов inttostr
end;

const
   ErrorHandler: TErrorProc = DefaultError;
```

В дальнейшем в программе использование вызова `ErrorHandler(I)` будет всегда эквивалентно выполнению процедуры `DefaultError`.

Процедурные переменные. Как только процедурный тип определен, можно объявлять переменные этого типа. Такие переменные называются процедурными переменными. Например, используя данное выше объявление типа, можно описать следующие переменные:

```
var
   P: TSwapProc;
   F: TMathFunc;
```

Подобно тому, как целочисленной переменной можно присвоить целочисленное значение, процедурной переменной можно присвоить процедурное значение.

Такое значение может, конечно, быть и другой процедурной переменной, но может также быть и идентификатором процедуры или функции. В этой ситуации объявление процедуры или функции можно рассматривать как особый вид объявления константы, значением которой является процедура или функция. Например, даны следующие объявления процедур и функций:

```
procedure Swap(var A, B: Integer);
var
   Tmp: Integer;
begin
   Tmp := A;
   A := B;
   B := Temp;
end;

function SqrTan(Angle: Real): Real;
begin
   SqrTan := Sqr(Tan(Angle));
end;
```

Объявленным ранее переменным `P` и `F` можно присвоить значения:

```
P := Swap;
F := SqrTan;
```

Согласно этим операторам присваивания, вызов `P(I,J)` эквивалентен вызову `Swap(I,J)`, а вызов `F(X)` — вызову `SqrTan(X)`.

Чтобы считаться совместимыми по присваиванию (т. е. использоваться в левой и правой части оператора присваива-

ния), процедурные типы должны отвечать следующим требованиям:

- иметь одинаковое число параметров;
- параметры в соответствующих позициях должны быть тождественных типов;
- типы результатов функций должны быть идентичны.

Как отмечено выше, названия параметров не имеют значения, когда проверяется совместимость типа процедуры.

В правой части оператора присваивания для процедурной переменной не могут находиться стандартные процедуры и функции и вложенные процедуры и функции.

Чтобы использовать стандартную процедуру или функцию в правой части оператора присваивания, ее необходимо дополнить внешней процедурой. Например, задан процедурный тип:

```
type
   StrProc = procedure(S: string);
```

Далее приведена совместимая по присваиванию процедура для вывода сообщения:

```
procedure ShowString (S: string);
begin
   ShowMessage(S);    // Стандартная процедура ShowMessage
end;
```

Вложенные процедуры и функции также нельзя использовать в правой части оператора присваивания. Процедура или функция называется вложенной, если она объявлена внутри другой процедуры или функции. В следующем примере функция `InnerProc` вложена в `OuterProc`, следовательно, значение `InnerProc` нельзя присвоить процедурной переменной.

```
procedure OuterProc;
var
   S: string;
function InnerProc(writestr:string):string;
begin
   ShowMessage(writestr+' - InnerProc ');
end;
begin
   S:= 'Вложенная процедура';
   ShowMessage(InnerProc(S));
end;
```

Использование процедурного типа не ограничивается процедурными переменными. Так же, как и любой другой тип, процедурный тип может присутствовать в объявлении структурного типа, например:

```
type
   TProc = procedure (X, Y: Integer; CurType:Byte);
   TProcList = array[1..10] of TProc;
   TWindow = record
           Header: string[31];
           Top, Left, Bottom, Right: Integer;
           SetCursor: TProc;
         end;
var
   PList : TProcList;
   W : TWindow;
```

Согласно этим объявлениям, следующие операторы являются вызовами процедур:

```
P[3] (1,1,3);
W.SetCursor(10,10,1);
```

Параметры процедурного типа. Так как процедурные типы допустимы в любом контексте, то можно объявить процедуры или функции, параметрами которых являются тоже процедуры или функции. Такие параметры являются параметрами-значениями и записываются в объявлении без служебного слова `var`. В качестве фактических параметров в этом случае используются объявленные процедуры или функции, имеющие соответствующее количество параметров требуемых типов.

Рассмотрим задачу формирования матрицы (двумерного массива) `M[i,j]`, элементами которой являются:

- сумма индексов `i` и `j`;
- произведение индексов `i` и `j`;
- сумма квадратов индексов `i` и `j`.

Пусть раздел объявлений программы, реализующей поставленную задачу, содержит описания следующих типов, переменных, функций и процедур:

```
type
   TFunc = function (X, Y: Integer): Integer;
                    // Объявление процедурного типа
   TMatrix = array [1..20, 1..20] of Integer;
                    // Объявление типа для матрицы
```

```
var
   M_Sum, M_Mult, M_SqrSum: TMatrix;
                        // Функция суммы двух чисел
function Sum(X,Y: Integer):Integer;
begin
   Sum := X + Y;
end;
                        // Функция произведения двух чисел
function Mult(X,Y:Integer):Integer;
begin
   Multiply := X * Y;
end;
                        //Функция суммы квадратов двух чисел
function SqrSum(X,Y: Integer): Integer;
begin
   SqrSum := (X * X) + (Y * Y);
end;
//Процедура заполнения матрицы с параметром процедурного типа
procedure FillMatrix(M:TMatrix; Operation:TFunc);
var
   I, J: Integer;
begin
   for I := 1 to 20 do   // Вложенные циклы заполнения матрицы
    for J := I to 20 do
         // Элемент матрицы вычисляется в зависимости
         // от используемой функции
      begin M[I,J] := Operation(I,J);
         M[J,I] := M[I,J];
      end
end;
```

Тогда поставленная задача решается с помощью следующих операторов:

```
FillMatrix(M_Sum, Sum);  // Элемент матрицы – сумма
                         // индексов i и j;
FillMatrix(M_Mult, Mult);
         // Элемент матрицы – произведение индексов i и j;
FillMatrix(M_SqrSum, SqrSum);
         // Элемент матрицы – сумма квадратов индексов i и j.
```

Параметры процедурного типа особенно полезны в ситуациях, когда над множеством процедур или функций выполняются общие действия. В нашем случае процедура `FillMatrix` представляет общее действие, выполняемое над функциями `Sum`, `Mult` и `SqrSum`.

4.1.9. Структура программы

Головной файл программы

Программа на языке Object Pascal представляет собой последовательность заголовка, раздела объявлений и тела программы.

Раздел объявлений включает в себя

- объявление меток;
- объявление констант;
- объявление типов;
- объявление переменных;
- объявление процедур и функций;
- объявление используемых модулей.

Порядок размещения разделов объявлений произвольный и может быть указано несколько одинаковых разделов объявлений.

Заголовок программы содержит служебное слово `program` и имя программы. Синтаксис заголовка следующий:

```
program <Имя программы>;
```

Заголовок программы является чисто декоративной деталью и компилятором не используется.

Синтаксис *раздела объявления меток* следующий:

```
Label <Список идентификаторов меток>;
```

Например:

```
Label 1, A1, 99;
```

Синтаксис *раздела объявления констант* следующий:

```
const <Список объявлений констант>;
```

В список объявлений констант могут входить объявления как простых, так и типизированных констант, например:

```
const
  CInt = 999;
  OddDigits : set of 0..9 = [1, 3, 5, 7, 9];
  RArr : array [1..5] of Real = (1.2, 0.5, -3.0, 75.12,
                                 123.675);
```

Синтаксис *раздела объявления типов* следующий:

```
type <Список объявлений типов>;
```

Список объявлений типов содержит перечень синтаксических конструкций вида:

<Идентификатор> = <Тип>

Например:

```
type
   TPoint = record
            X, Y, Z: Real
        end;
   TArrPoint = array [1..1000] of TPoint;
   TFilePoint = file of TPoint;
```

Синтаксис *раздела объявления переменных* следующий:

var <Список объявлений переменных>;

Например:

```
var
   PointA, PointB: TPoint;
   i, j, k: Integer;
   Smess: string;
   ChrMess: set of 'A'..'Z';
```

Синтаксис *раздела объявления процедур и функций* следующий:

<Список объявлений подпрограмм>;

Например:

```
          // Функция вычисления факториала натурального числа
function Factorial(N:integer): Integer;
var i, F: Integer;
begin
   F := 1;
   for i := 2 to N do F := F*i;
   Factorial := F
end;
          // Процедура удаления пробелов из строки
procedure DeleteBlank(var S:string);
var i: Integer;
begin
   i := pos(' ',S);
   while i > 0 do
       begin  Delete(S, i, 1);
              i := pos' ',S)
       end;
end;
```

Предложение uses идентифицирует все модули, используемые программой. Синтаксис *раздела объявления используемых модулей* следующий:

```
uses <Список модулей>;
```

Например:

```
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs, StdCtrls;
```

Тело программы представляет собой составной оператор — начинается словом begin и заканчивается словом end с точкой. Точка является признаком конца программы.

Модули

Модули являются основой модульного программирования. Они используются для создания библиотек, которые могут включаться в различные программы (при этом становится необязательным иметь в наличии исходный код), а большие программы могут подразделяться на логически связанные модули.

Общий синтаксис модулей:

```
<заголовок модуля >;
<интерфейсный раздел>
<раздел реализации>
<раздел инициализации>.
```

Заголовок модуля определяет имя модуля:

```
unit <имя модуля >
```

Имя модуля используется при ссылке на модуль в предложении uses. Это имя должно быть уникальным: два модуля с одним именем не могут использоваться одновременно.

Интерфейсный раздел объявляет те константы, типы, переменные, процедуры и функции, которые являются глобальными, то есть доступными основной программе (программе или модулю, которые используют данный модуль):

```
interface
<предложение uses >;
<раздел объявления констант >;
<раздел объявления типов >;
<раздел объявления переменных >;
<раздел объявления процедур и функций >;
```

Интерфейсный раздел только перечисляет заголовки процедур и функций. Полные описания процедур и функций находятся в разделе реализации.

Раздел реализации содержит описания всех глобальных процедур и функций. В нем также описываются константы, типы, переменные, процедуры и функции, являющиеся глобальными для модуля, но локальными по отношению к основной программе:

```
implementation
<предложение uses >;
<раздел объявления меток >;
<раздел объявления констант >;
<раздел объявления типов >;
<раздел объявления переменных >;
<раздел объявления процедур и функций >;
```

Заголовки процедур и функций могут быть продублированы из интерфейсного раздела. Необязательно задавать список формальных параметров, но если список указан, то в случае несоответствия объявления в интерфейсном разделе и разделе реализации компилятор выдаст ошибку.

Раздел инициализации является последним разделом модуля. Он может состоять либо из зарезервированного слова `end` (в этом случае модуль не содержит кода инициализации), либо представлять собой составной оператор, который должен выполняться при инициализации модуля.

Разделы инициализации модулей, используемых программой, выполняются в том же порядке, в каком модули указаны в предложении `uses`.

4.1.10. Организация ввода-вывода данных. Работа с файлами

Поддержка работы с внешними файлами в Delphi осуществляется несколькими принципиально различными (с точки зрения программиста) способами:

- через файловые переменные;
- через дескрипторы файлов;
- через использование библиотечных компонентов.

Работа с файловыми переменными

Связь с внешними файлами осуществляется через файловые переменные, т. е. переменные файлового типа. Object Pascal обеспечивает доступ к трем различным категориям файлов:

- текстовые файлы (для связи с такими файлами необходимо объявить переменную типа `Text` или `TextFile`);
- типизированные файлы (для связи с ними объявляется переменная типа `file of <Тип>`);
- нетипизированные файлы или файлы без типа (связываются с программой через переменную типа `file`).

Работа с каждой из категорий имеет свою специфику, но есть и общие правила обеспечения процессов чтения-записи для всех категорий файлов.

Работу с внешними файлами поддерживают стандартные процедуры и функции ввода-вывода. Перед использованием файловой переменной любого типа ее необходимо связать с внешним файлом с помощью вызова процедуры `AssignFile`. После того, как связь с внешним файлом установлена, все операции ввода или вывода для внешнего файла осуществляются через присоединенную к нему файловую переменную.

Перед непосредственным чтением-записью данных внешний файл необходимо «открыть». Существующий файл можно открыть с помощью процедуры `Reset`, а новый файл можно создать и открыть с помощью процедуры `Rewrite`. Кроме того, текстовые файлы могут быть открыты процедурой `Append` для добавления данных в конец файла, но при этом они доступны только для записи.

Типизированные и нетипизированные файлы всегда допускают как чтение, так и запись независимо от того, были они открыты с помощью процедуры `Reset` или с помощью процедуры `Rewrite`.

Любой файл логически представляет собой линейную последовательность элементов (записей). Каждая запись файла имеет свой порядковый номер. Первая запись файла считается нулевой и имеет порядковый номер 0. Для каждого файла определено понятие текущей позиции внутри файла, т. е. порядковый номер записи (или записей), к которой будут обращены стандартные процедуры чтения-записи.

Обычно доступ к файлам организуется последовательно, т. е. когда некоторая запись считывается с помощью стандартной про-

цедуры Read или записывается с помощью стандартной процедуры Write, текущая позиция файла перемещается к следующей по порядку записи файла. Однако к типизированным и нетипизированным файлам можно организовать прямой доступ с помощью стандартной процедуры Seek, которая перемещает текущую позицию файла к записи с заданным порядковым номером. Текущую позицию в файле и текущий размер файла можно определить с помощью стандартных функций FilePos и FileSize.

Когда программа завершит обработку файла, его необходимо закрыть с помощью стандартной процедуры CloseFile. После завершения процедуры связанный с файловой переменной внешний файл обновляется, а файловая переменная может быть использована для обеспечения работы с другим внешним файлом.

По умолчанию при всех обращениях к стандартным функциям и процедурам ввода-вывода автоматически производится проверка на наличие ошибок. При обнаружении ошибки программа прекращает работу и выводит на экран сообщение об ошибке. С помощью директив компилятора {$I+} и {$I-} эту автоматическую проверку можно включить или выключить. Когда автоматическая проверка отключена (т. е. когда процедура или функция была скомпилирована с директивой {$I-}), ошибки ввода-вывода, возникающие при работе программы, не приводят к ее останову. Результат выполнения операции ввода-вывода при этом можно проверить с помощью стандартной функции IOResult.

Текстовые файлы. При открытии текстового файла внешний файл интерпретируется особым образом: считается, что он представляет собой последовательность символов, сгруппированных в строки, где каждая строка заканчивается признаком конца строки (end of line). Признак конца строки представлен символом перевода каретки, за которым, возможно, следует символ перевода строки.

Для текстовых файлов существует специальный вид операций чтения и записи (Read и Write), которые позволяют считывать и записывать последовательности значений, тип которых отличается от типа Char и string. Такие значения автоматически переводятся в символьное представление и обратно. Рассмотрим, например, оператор процедуры

```
Read(F, i, R);
```

где i — переменная типа Integer, а R — переменная типа Real. Использование этой процедуры приведет к считыванию двух по-

следовательностей цифр и знаков, допустимых при записи чисел. Эти последовательности в файле должны быть разделены пробелом, знаком табуляции или признаком конца строки. Первая последовательность будет интерпретирована как десятичное число, значение которого будет занесено в переменную i, а вторая последовательность будет преобразована в вещественное число, которое будет занесено в переменную R.

С каждой файловой переменной связана запись типа TTextRec, содержащая информацию, необходимую для работы с файлом файловой системы. Например, выражение

```
TTextRec(F).Name
```

в качестве значения выдаст имя файла, связанного в данный момент с файловой переменной F.

Типизированные файлы. Использование стандартных процедур Read и Write для типизированных файлов отличается от их использования для текстовых файлов тем, что переменные, в которые читается или из которых записывается информация, должны иметь тот же тип данных, что и компоненты типизированного файла. Таким образом, при чтении или записи типизированных файлов всегда происходит перемещение данных одинаковой длины.

Функция FileSize для типизированных файлов возвращает количество компонентов, записанных в файл, а функция FilePos позволяет определить номер текущей компоненты файла.

Нетипизированные файлы. Нетипизированные файловые переменные используются в основном для обеспечения прямого доступа к любому файлу на диске независимо от его типа и структуры.

Любой нетипизированный файл объявляется со словом file без атрибутов, например:

```
var
   Datafile : file;
```

Для нетипизированных файлов в процедурах Reset и Rewrite допускается указывать дополнительный параметр, чтобы задать размер записи, использующийся при передаче данных.

По умолчанию длина записи равна 128 байт. Предпочтительной длиной записи является длина записи, равная 1, поскольку

это единственное значение, которое точно отражает размер любого файла (если длина записи равна 1, то неполные записи невозможны).

За исключением процедур `Read` и `Write` для всех нетипизированных файлов допускается использование любой стандартной процедуры, которую разрешено использовать с типированными файлами. Вместо процедур `Read` и `Write` здесь используются соответственно процедуры `BlockRead` и `BlockWrite`, позволяющие пересылать данные с высокой скоростью.

Стандартные процедуры и функции ввода-вывода

В табл. 4.15 приведено описание стандартных процедур и функций для работы с файлами через файловые переменные.

Таблица 4.15. **Стандартные процедуры и функции ввода-вывода**

Наименование	Статус	Описание
`AssignFile(F,Name)`	P	Присваивает имя внешнего файла (`Name`) файловой переменной `F`. Если строка имени пустая, осуществляется связь со стандартным файлом ввода или вывода
`Append(F)`	P	Открывает существующий текстовый файл только для дозаписи, внутренний файловый указатель устанавливается на конец файла
`BlockRead (F,Buf,N[,NTransf])`	P	Читает `N` или менее (если при чтении достигнут конец файла) записей из нетипизированного файла, с которым связана файловая переменная `F`. Результат чтения помещается в переменную `Buf`. Необязательный параметр `NTransf` указывает на количество фактически прочитанных записей. Максимальный размер считываемых данных равен `Size*N`, где `Size` — размер записи, задающийся процедурами `Reset` и `Rewrite`. После выполнения операции внутренний указатель файла перемещается на количество считанных записей (`NTransf`)
`BlockWrite (F,Buf, N[,NTransf])`	P	Записывает `N` или менее (если при записи будет до конца заполнен диск) записей в нетипизированный файл, с которым связана файловая переменная `F`. Данные для записи берутся из переменной `Buf`. Необязательный параметр `NTransf` указывает на количество фактически перемещенных записей.

Продолжение табл. 4.15

Наименование	Статус	Описание
		Максимальный размер перемещаемых данных равен `Size*N`, где `Size` — размер записи, задающийся процедурами `Reset` и `Rewrite`. После выполнения операции внутренний указатель файла перемещается на количество помещенных в файл записей (`NTransf`)
`ChDir(SPath)`	P	Меняет текущую директорию на директорию с именем, задающимся параметром `SPath`. Если параметр `SPath` содержит имя диска, то меняется и текущий диск
`CloseFile(F)`	P	Закрывает ранее открытый внешний файл, с которым связана файловая переменная `F`
`Eof(F)`	F	Возвращает статус конца файла, с которым связана файловая переменная `F`. Если параметр опущен, рассматривается стандартный файл ввода. Принимает значение `True`, если обработана последняя запись файла или файл является пустым. В остальных случаях принимает значение `False`
`Eoln(F)`	F	Возвращает статус конца строки для текстового файла, с которым связана файловая переменная `F`. Если параметр опущен, рассматривается стандартный файл ввода. Принимает значение `True`, если текущим элементом файла является признак конца строки или `Eof(F)` принимает значение `True`. В остальных случаях принимает значение `False`
`Erase(F)`	P	Удаляет внешний файл, с которым связана файловая переменная `F`. Перед удалением файл обязательно должен быть закрыт с помощью процедуры `CloseFile`
`FilePos(F)`	F	Возвращает текущую позицию (в количестве записей) указателя внутри типизированного или нетипизированного файла, с которым связана файловая переменная `F`. Если указатель находится в начале файла, возвращает значение 0. Применяется только к открытым файлам
`FileSize(F)`	F	Возвращает текущий размер (в количестве записей) типизированного или нетипизированного файла, с которым связана файловая переменная `F`. Применяется только к открытым файлам. Для пустых файлов возвращает значение 0

Продолжение табл. 4.15

Наименование	Статус	Описание
`Flush(F)`	P	Освобождает буфер текстового файла вывода. Информация из буфера записывается во внешний текстовый файл, с которым связана файловая переменная `F`. Не работает для файлов, открытых на чтение
`GetDir(D, CPath)`	F	Возвращает в значение параметра `CPath` текущую директорию указанного устройства. Номер устройства задается параметром `D`: `D=0` — текущий диск, `D=1` — диск `A`, `D=2` — диск `B` и т. д.
`IOResult`	F	Возвращает целое значение статуса последней операции ввода-вывода
`MkDir(Path)`	P	Создает новую директорию, путь к которой задается параметром `Path`. В строку `Path` не должно входить имя файла
`Read(F,v1[,v2,...,vn])`	P	Читает одно или более значений из файла в одну или более переменных. Используется для текстовых и типизированных файлов. Параметр `F` может отсутствовать. Связь в этом случае осуществляется со стандартным текстовым файлом ввода
`Readln(F,v1[,v2,...,vn])`	P	Читает одно или более значений из текстового файла в одну или более переменных и переводит текущий указатель файла на начало следующей строки. Используется только для текстовых файлов. Параметр `F` может отсутствовать. Связь в этом случае осуществляется со стандартным текстовым файлом ввода. Вызов процедуры в форме `Readln(F)` переводит внутренний указатель файла на начало следующей строки (или на конец файла, если следующей строки не существует)
`Rename(F,NewNm)`	P	Переименовывает внешний файл, с которым связана файловая переменная `F`. Параметр `NewNm` содержит новое имя файла
`Reset(F[,Size])`	P	Открывает на чтение и запись существующий файл, с которым связана файловая переменная `F`. Внутренний указатель файла устанавливается на начало файла. Если строка имени файла пустая (`AssignFile(F,'')`), осуществляется связь со стандартным файлом ввода. Если файловая переменная `F` имеет тип `text`, то файл открывается только на чтение.

Продолжение табл. 4.15

Наименование	Статус	Описание
		После вызова процедуры `Reset` значение функции `Eof(F)` всегда `False` (за исключением случая пустого файла — тогда `Eof(F)=True`). Необязательный параметр `Size` (тип — целое) используется только для файлов без типа и задает размер пересылаемой записи в байтах. По умолчанию `Size` = 128
`Rewrite(F[,Size])`	P	Создает и открывает на чтение и запись новый файл, имя которого задается процедурой `AssignFile` для файловой переменной `F`. Если файл с таким именем уже существует, то он удаляется и на его месте создается новый пустой файл. Внутренний указатель файла устанавливается на начало пустого файла. Если строка имени файла пустая (`AssignFile(F,'')`), осуществляется связь со стандартным файлом вывода. Если файловая переменная `F` имеет тип `text`, то файл открывается только на запись. После вызова процедуры `Rewrite` значение функции `Eof(F)` всегда `True`. Необязательный параметр `Size` (тип — целое) используется только для файлов без типа и задает размер пересылаемой записи в байтах. По умолчанию `Size` = 128
`RmDir(SPath)`	P	Удаляет пустую директорию, путь к которой указан параметром `SPath`. Если указан путь к несуществующей директории, не пустой директории или к директории, установленной как текущая, выдается сообщение об ошибке
`Seek(F, N)`	P	Устанавливает текущую позицию указателя внутри типизированного или нетипизированного файла, с которым связана файловая переменная `F`, на запись с номером `N`. Нумерация записей ведется со значения 0. Применяется только к открытым файлам. `Seek(F, FileSize(F))` перемещает внутренний указатель на конец файла
`SeekEof(F)`	F	Возвращает статус конца текстового файла, с которым связана файловая переменная `F`. Если параметр опущен, рассматривается стандартный файл ввода. Отличается от `Eof(F)` тем, что стоящие в конце файла символы пробела и табуляции игнорируются

Окончание табл. 4.15

Наименование	Статус	Описание
`SeekEoln(F)`	F	Возвращает статус конца строки для текстового файла, с которым связана файловая переменная `F`. Если параметр опущен, рассматривается стандартный файл ввода. Отличается от `Eoln(F)` тем, что стоящие в конце строки символы пробела и табуляции игнорируются
`SetTextBuf (F,Buf[,Size])`	P	Назначает буфер ввода-вывода для внешнего текстового файла, с которым связана файловая переменная `F`. Параметр `Buf` указывает на буфер, который будет использоваться для ввода-вывода. Необязательный параметр `Size` задает размер буфера. Если размер не указан, используется вся переменная `Buf`. Если процедура не используется, ввод-вывод организуется с помощью стандартного буфера размером в 128 байт. Назначение действует до следующей процедуры `AssignFile`. Процедура не должна применяться к уже открытому файлу, для которого имели место операции ввода-вывода (это может привести к потере данных)
`Truncate(F)`	P	Удаляет часть файла, с которым связана файловая переменная `F`, начиная с текущей позиции до конца. Используется только для типизированных и нетипизированных файлов. Применяется только к открытым файлам
`Write(F,v1[,v2,..., vn])`	P	Записывает одно или более значений в файл, с которым связана файловая переменная `F`. Используется для текстовых и типизированных файлов. Параметр `F` может отсутствовать. Связь в этом случае осуществляется со стандартным текстовым файлом ввода
`Writeln(F,v1[,v2,...,vn])`	P	Записывает одно или более значений в файл, с которым связана файловая переменная `F`, и затем записывает признак конца строки (только для текстовых файлов). Используется только для текстовых файлов. Параметр `F` может отсутствовать. Связь в этом случае осуществляется со стандартным текстовым файлом вывода. Вызов процедуры в форме `Writeln(F)` приводит к записи в файл признака конца строки

4.2. Интегрированная среда разработки приложений Delphi

При описании среды разработки приложений Delphi используется версия программного продукта Delphi 7.

4.2.1. Интерфейс среды Delphi

В момент первой загрузки среды Delphi интерфейс первоначально включает шесть окон (рис. 4.1):

- Главное окно Delphi с заголовком, содержащим имя открытого проекта (на рисунке — **Delphi7 — Project2**);
- окно Обозревателя дерева объектов (**Object Tree View**);
- окно Инспектора объектов (**Object Inspector**);
- окно Конструктора формы с заголовком **Form1**;
- окно Редактора кода, озаглавленное как **Unit1.pas**, которое первоначально перекрывается окном формы;
- окно Проводника кода (**Exploring Unit1.pas**).

Окна Delphi можно перемещать, убирать с экрана, а также изменять их размеры.

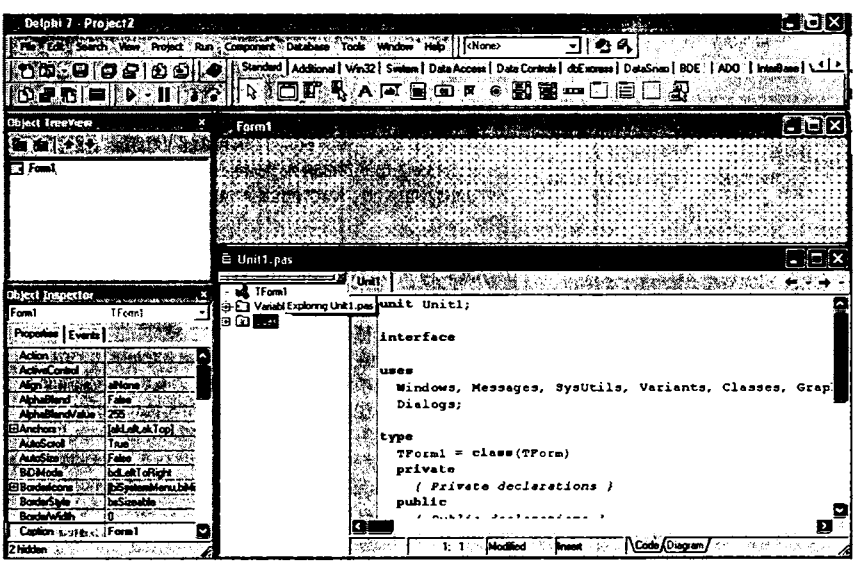


Рис. 4.1. Исходный вид Delphi

Delphi — однодокументная среда, позволяющая одновременно работать только с одним приложением.

Главное окно

Главное окно Delphi, занимающее верхнюю часть экрана, содержит:

- строку заголовка, в которой отображается имя открытого проекта;
- строку Главного меню с набором команд для управления разработкой и тестированием приложений;
- панель инструментов — кнопок, представляющих собой краткую форму функций меню;
- палитру компонентов, отображающую компоненты, с помощью которых создается приложение.

Панель инструментов находится ниже меню в левой части экрана, а палитра компонентов — ниже меню и справа от панели инструментов.

Панель инструментов объединяет кнопки для вызова наиболее часто используемых команд Главного меню. Вызвать команды Главного меню можно также с помощью соответствующих комбинаций клавиш алфавитно-цифровой и функциональной клавиатур.

Кнопки сгруппированы в шесть панелей инструментов:

- стандартную;
- просмотра;
- отладки;
- пользовательскую;
- Рабочий стол;
- Internet.

Управление отображением панелей инструментов и изменением состава кнопок на них осуществляется с помощью контекстного меню панелей инструментов или с помощью функции Главного меню **View → ToolBars → Customize...** .

Палитра компонентов используется для выбора и размещения на форме компонентов графического интерфейса. Компоненты являются «строительными блоками», из которых конструируются формы приложений. Группы компонентов размещаются на разных вкладках.

В Delphi используется *открытая* компонентная архитектура, которая позволяет добавлять компоненты в каждую группу и создавать новые группы компонентов, таким образом, разные

конфигурации среды могут отличаться набором представленных компонентов.

Палитру компонентов можно настраивать с помощью диалогового окна **Pallet Properties** (свойства палитры), которое вызывается командой **Properties** (свойства) контекстного меню Палитры компонентов или командой **Component → Configure Palette** Главного меню. Окно позволяет выполнять операции по удалению, добавлению и перемещению отдельных компонентов и целых страниц (рис. 4.2).

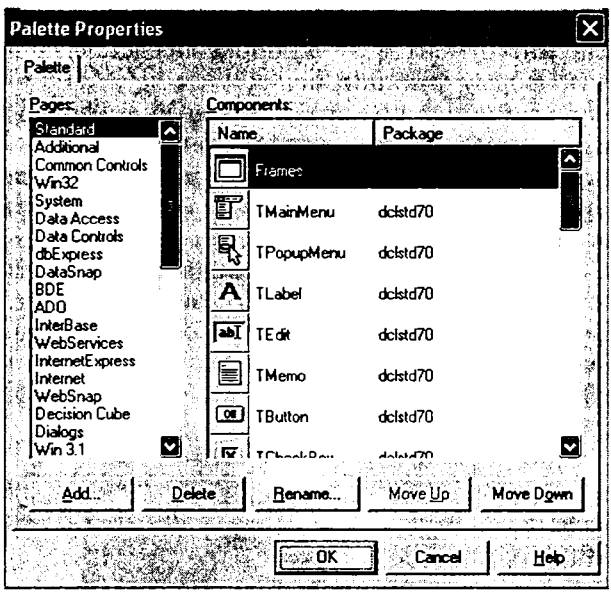


Рис. 4.2. Диалоговое окно свойств палитры компонентов

Окно Обозревателя дерева объектов

Обозреватель дерева объектов (**Object Tree View**) после запуска среды находится под главным окном и отображает древовидную структуру объектов текущей формы.

Окно Инспектора объектов

Окно Инспектора объектов (**Object Inspector**) предназначено для управления компонентами (их размещением и поведением

на форме) на этапе проектирования интерфейса. Оно содержит две страницы — «Свойства» (**Properties**) и «События» (**Events**). Каждый компонент имеет свой набор свойств и событий, определяющий его индивидуальные характеристики и особенности, а также возможности по его использованию в процессе работы приложения.

Страница **Properties** (список свойств) отображает и позволяет менять характеристики текущего (выделенного) объекта в окне формы. Когда в среде Delphi создается новое приложение, первоначально Инспектор объектов отображает свойства текущей формы (см. рис. 4.1). Если при проектировании интерфейса нужно изменить что-нибудь, связанное с определенным компонентом, то это выполняется в Инспекторе объектов. К примеру, можно изменить заголовок и размер компонента `TLabel`, изменяя свойства `Caption`, `Height` и `Width` (рис. 4.3).

На странице **Events** (список событий) задаются действия, которые должны выполняться объектом при наступлении определенных событий. Страница событий связана с Редактором кода: если курсор манипулятора или текстовый курсор расположен в некоторой строке на правой половине страницы, то при двойном нажатии клавиши манипулятора «мышь» автоматически в

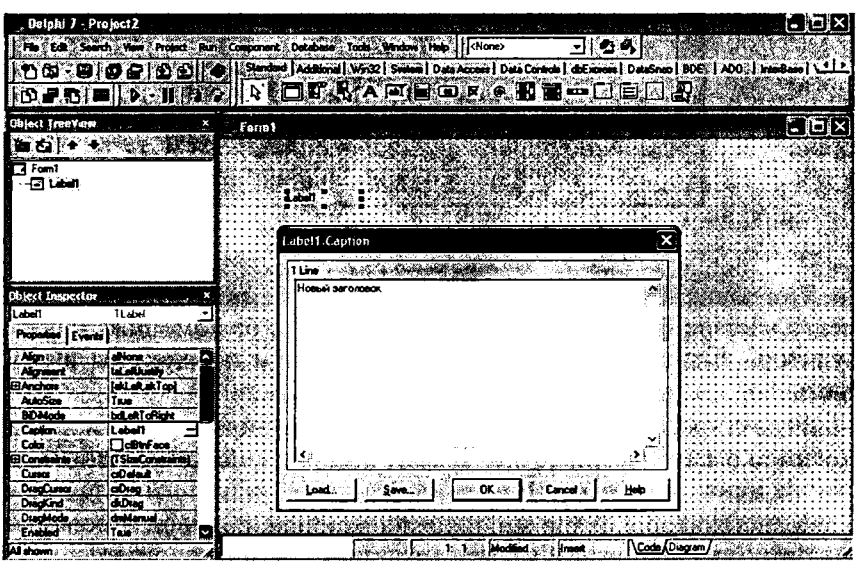


Рис. 4.3. Изменение свойства Caption для TLabel

окне Редактора кода создается заготовка для процедуры обработки соответствующего события.

Если для какого-либо события объявлена процедура, то в дальнейшем при работе с приложением она выполняется автоматически при возникновении этого события. Такие процедуры называют *обработчиками*, так как они служат для обработки событий.

Окно Конструктора формы

Конструктор формы — это окно разрабатываемого приложения, внешний вид и наполнение которого определяется при проектировании. Компоненты графического интерфейса помещаются на форму и располагаются на ней по желанию проектировщика (рис. 4.4). Приложение может иметь несколько форм.

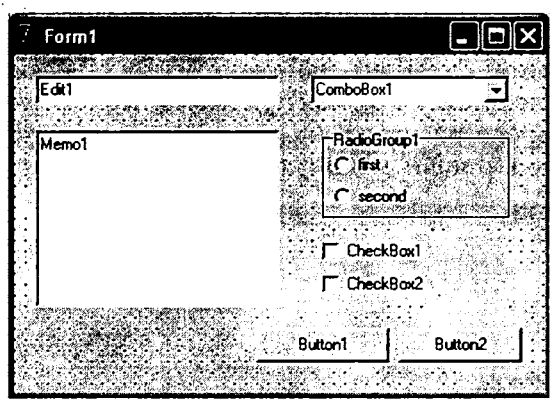


Рис. 4.4. Окно формы

Первоначально форма имеет заголовок **Form1**. Объекты, размещаемые на форме, должны быть выбраны (с помощью манипулятора «мышь») на Палитре компонентов и перенесены в область формы (путем указания места расположения).

Редактировать размещение компонента можно с помощью группы команд Главного меню **Edit**, а также с помощью контекстного меню.

Конструктор формы интуитивно понятен и прост в использовании, что в большой степени упрощает создание визуального интерфейса.

Окно редактора кода

Окно редактора кода (рис. 4.5) предназначено для редактирования исходного кода модуля программы, описывающего данную форму. Это обычный текстовый редактор, с помощью которого можно редактировать текст модуля и другие текстовые файлы приложения (а также и простые текстовые файлы в кодировке Windows). В редакторе кода можно открывать несколько файлов, каждый из которых размещается на отдельной странице.

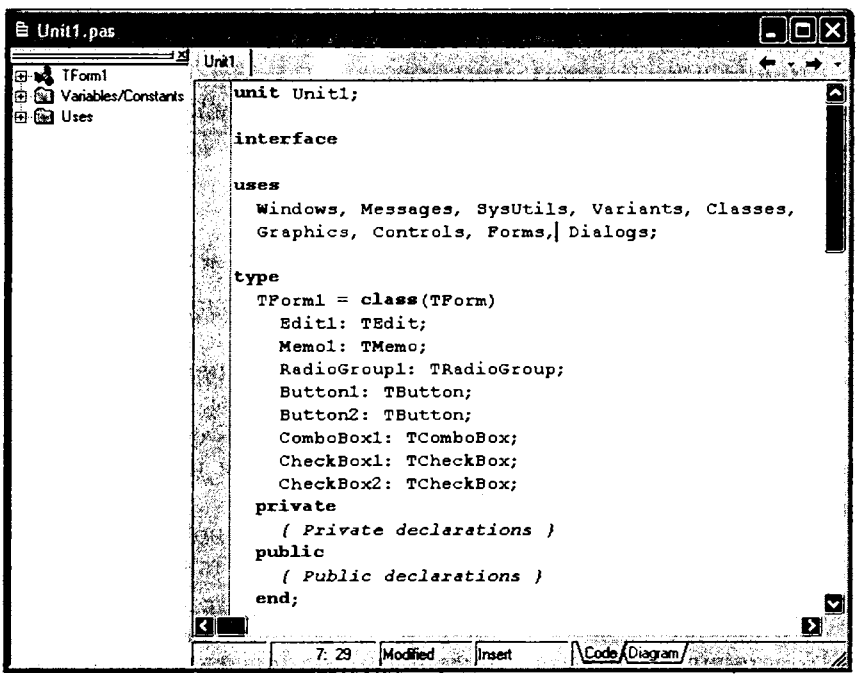


Рис. 4.5. Окно редактора кода

Окно редактора кода первоначально размещено «под» окном формы, поскольку первый этап разработки формы — размещение на ней элементов графического интерфейса — сопровождается автоматическим генерированием программного кода. Необходимость использования окна модуля для ввода и редактирования наступает, когда программируются обработчики событий для размещенных на форме компонентов.

Переключение между редактором кода и формой поддерживается функциональной клавишей <**F12**>.

Окно Проводника кода

Окно Проводника кода располагается в левой части окна редактора кода. В нем в виде дерева отображаются все объекты модуля формы (например, переменные и процедуры). В окне Проводника кода удобно просматривать объекты приложения и обращаться к ним, что особенно важно при работе с большими модулями. В функции Проводника входит и автоматизированное создание новых классов.

При закрытии файла закрывается и Проводник кода. Проводник кода можно убирать и отображать с помощью команды меню **View → Code Explorer**.

4.2.2. Характеристика проекта Delphi

Любой проект представляет собой совокупность не менее чем семи файлов:

- главный файл проекта — файл с расширением `.dpr`, представляет собой основной модуль программы;
- файл главной формы (описания формы) — файл с расширением `.dfm`, используется для сохранения информации о внешнем виде главной формы;
- первый модуль программы (модуль главной формы) — файл с расширением `.pas`, автоматически появляется вначале работы;
- файл ресурсов — файл с расширением `.res`. Содержит иконку для проекта, создается автоматически и имеет то же имя, что и главный файл проекта;
- файл параметров проекта — файл с расширением `.cfg`, текстовый файл для сохранения конфигурации данного проекта. Имя файла совпадает с именем главного файла проекта;
- файл параметров среды — файл с расширением `.dof`, текстовый файл, в котором хранятся текущие установки параметров проекта, таких, как параметры компиляции, рабочие директории, условные директивы, параметры команд-

ной строки. Имя файла совпадает с именем главного файла проекта;

- файл настроек рабочей области среды (Desktop File) — файл с расширением .dsk, в котором сохраняется состояние среды Delphi для проекта. Имя файла совпадает с именем главного файла проекта.

Помимо перечисленных файлов в проект могут входить и дополнительные модули — файлы с расширением .pas (рис. 4.6).

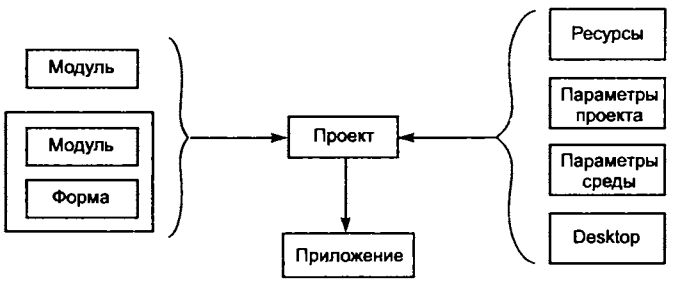


Рис. 4.6. Структурная схема приложения Delphi

Файлы проекта по умолчанию располагаются в одном каталоге.

Если главный файл проекта сохраняется под другим именем, то это имя получают и файлы с расширением .res, .dof, .cfg и .dsk.

При запуске Delphi автоматически создается новый проект с именем Project1.dpr, который имеет в своем составе форму Form1.dfm и соответствующий ей модуль Unit1.pas.

Главный файл проекта

При выполнении операций с проектом код файла проекта (программы) формируется средой Delphi автоматически:

```
program Project1;          // Имя программы

uses
          // Далее следует перечисление используемых модулей
  Forms,          // Имя подключаемого модуля
  Unit1 in 'Unit1.pas'  {Form1};
        // Перечисление модулей всех форм проекта

{$R *.RES}  // Директива подключения к проекту файл-ресурсов
```

```
begin          // Начало блока программы
 Application.Initialize;     // Инициализация приложения
 Application.CreateForm(TForm1,Form1);     // Создание формы
 Application.Run;        // Запуск приложения
end.           // Конец блока программы
```

Служебное слово `uses` сообщает компилятору, какие модули должны быть подключены при построении приложения. В приведенном примере подключается библиотечный модуль `Forms` и модуль с исходным кодом формы `Unit1.pas`. Имя формы (`Form1`) указано в виде комментария. Если проект содержит несколько форм, перечисляются модули всех форм проекта.

Просмотреть и отредактировать код файла в окне Редактора кода можно с помощью команды **Project → View Source**.

Подключаемый файл ресурсов имеет имя, совпадающее с именем файла проекта.

Файлы формы

Для каждой формы автоматически создаются файл описания, первоначально имеющий имя `Unit1.dfm`, и файл модуля `Unit1.pas` (файлы модуля формы и описания формы имеют всегда одинаковое, но отличающееся от имени файла проекта, имя).

Файл описания формы (*.dfm) — текстовый файл, содержащий параметры формы и ее компонентов. При конструировании формы в Файл описания автоматически вносятся соответствующие изменения.

Чтобы отобразить содержание этого файла на экране, необходимо активизировать команду контекстного меню Окна формы **View as Text** (или нажать функциональные клавиши <**Alt+F12**>).

Окно Редактора кода и его содержимое будут доступны для просмотра и редактирования описания формы (рис. 4.7).

Переключиться в режим формы можно с помощью команды контекстного меню **View As Form** или функциональных клавиш <**Alt+F12**>.

Чтобы открыть окно любой формы проекта для конструирования, необходимо выбрать ее в диалоговом окне, появляющемся по команде Гланого меню **View → Forms**.

Файл модуля формы (*.pas) создается автоматически при добавлении новой формы и содержит описание класса формы (состав компонентов и функционирование обработчиков событий).

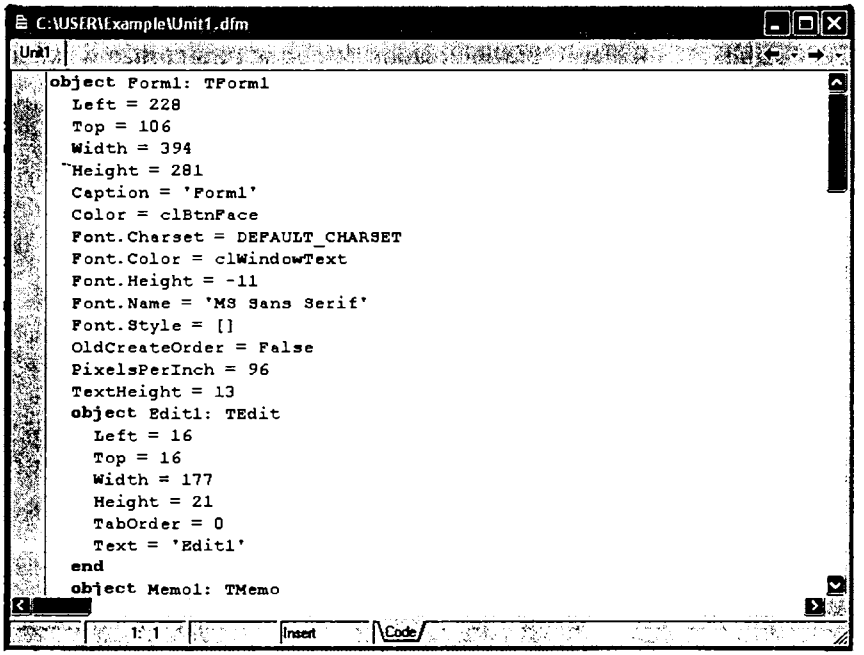


Рис. 4.7. Содержимое файла описания формы

В процессе конструирования формы (при размещении на форме компонентов) в модуль формы вносятся соответствующие изменения, причем изменения в описание класса вносятся *автоматически*, а процедуры обработки событий *кодируются разработчиком*.

Открыть модуль формы можно с помощью команды меню **File** → **Open** либо в диалоговом окне команды **View** → **Units**, выбрав нужный модуль (рис. 4.8).

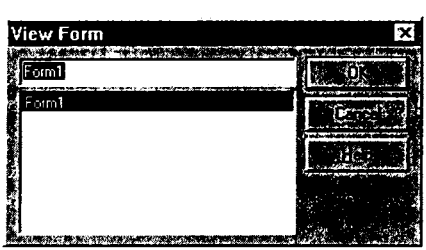


Рис. 4.8. Выбор формы для конструирования

Тексты модулей форм отображаются в окне Редактора кода и редактируются с его помощью.

Файлы модулей

Помимо файлов, создаваемых средой Delphi, в проект могут включаться файлы, не связанные с формами. Например, константы, переменные, процедуры и функции, общие для нескольких модулей проекта, целесообразно оформить в виде отдельного модуля и подключать его по мере необходимости. Такие модули оформляются по правилам языка программирования Object Pascal и сохраняются в отдельных файлах с расширением `.pas`.

Для того чтобы модуль в дальнейшем мог быть использован другим модулем или проектом, его имя должно быть указано в разделе `uses` этого модуля или проекта, как имя подключаемого модуля.

Файл ресурсов

При первом сохранении проекта автоматически создается файл ресурсов с именем, совпадающим с именем файла проекта, и расширением `.res`.

В файле ресурсы разбиты на группы. Каждая группа имеет имя, а каждый ресурс уникально поименован в пределах группы. Имя ресурса задается при его создании и в последующем используется в приложении для доступа к этому ресурсу.

Файл содержит следующие ресурсы:

- пиктограммы;
- растровые изображения;
- курсоры.

Первоначально Файл ресурсов содержит пиктограмму проекта. Ее можно изменить, используя редактор изображений (**Image Editor**). Вызывается редактор командой **Tools** → **Image Editor**.

Редактор изображений позволяет обрабатывать файлы четырех видов. Три из них объединяют файлы, содержащие ресурсы, указанные выше: пиктограммы приложений (`*.ico`), растровые изображения (`*.bmp`), курсоры (`*.cur`). К последнему (четвертому) виду относятся файлы, имеющие формат откомпилированных файлов-ресурсов (файлы с расширением `.res`), которые могут в свою очередь содержать ресурсы предыдущих трех видов.

На рис. 4.9 показано окно редактора, в которое загружен файл, и выполняется редактирование пиктограммы приложения.

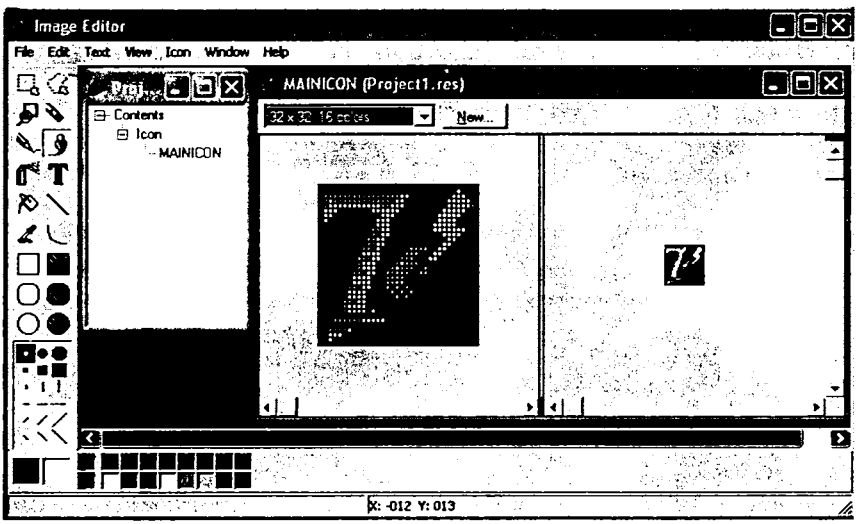


Рис. 4.9. Пиктограмма проекта в окне Редактора изображений

Пиктограмма проекта находится в группе **Icon** и по умолчанию имеет имя `MAINICON`.

Файл параметров проекта

Файл параметров проекта содержит использующиеся при компиляции приложения конфигурационные установки, такие как директории для поиска файлов проекта и текущие установки директив компиляции.

Например, если конфигурационный файл содержит следующие строки:

```
-$D+
-$I+
-U"C:\ DELPHI\UNITS"
```

это означает, что проект будет содержать коды отладчика (`$D+`), в проект будет включена автоматическая генерация результатов выполнения процедур ввода-вывода (`$I+`), и при построении приложения для поиска модулей будет использоваться директория `C:\DELPHI\UNITS`.

Для установки параметров проекта используются страницы **Forms** и **Application** окна параметров проекта (**Project Options**), которое открывается командой Главного меню **Project** → **Options** (рис. 4.10).

Рис. 4.10. Окно установки параметров проекта

Файл параметров среды

Файл параметров среды представляет собой текстовой файл, который содержит текущие установки для параметров проекта:

- настройки компилятора и компоновщика;
- имена служебных каталогов;
- директивы условной компиляции;
- параметры командной строки.

Для установки параметров проекта используется диалоговое окно, вызываемое с помощью команды меню **Project** → **Options.** Параметры разбиты на группы, каждая из которых располагается на соответствующей странице (см. рис. 4.10). После установки отдельных параметров Delphi автоматически вносит нужные изменения в файл параметров среды, представляя информацию в виде текстовых строк.

Например, к проекту на стадии разработки имеет смысл подключать отладочную информацию. Для этого необходимо установить опцию **Debug Information** на странице **Compiler**.

Файл настроек рабочей области среды

Файл содержит настройки рабочей области для текущего проекта, например, информацию о том, какие окна открыты, где находится курсор и т. п. Такая информация позволяет восстановить состояние рабочей области при каждом новом открытии проекта в среде.

Чтобы обеспечить автоматическое создание и сохранение файла настроек, необходимо:

1) с помощью команды меню **Tools** → **Environment Options** открыть диалоговое окно (рис. 4.11);

2) на странице **Preferences** окна в разделе **Autosave options** установить параметр **Project Desktop**.

Среда Delphi обновляет файл настроек рабочей области всякий раз при закрытии проекта. Файл хранится в той же директории, что и главный файл проекта, и имеет то же имя.

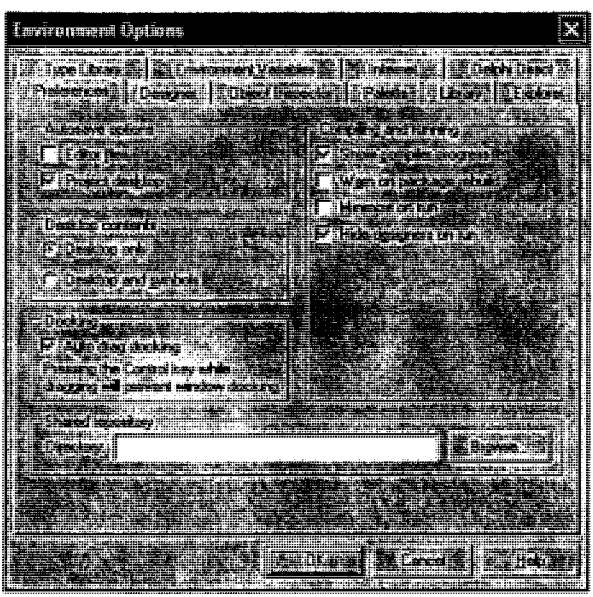


Рис. 4.11. Окно настроек среды

При новой загрузке проекта внешний вид среды восстанавливается, т. е. становится таким же, как при предыдущем закрытии проекта.

При создании нового проекта опция автоматического сохранения настроек проекта по умолчанию всегда включена.

Резервные файлы

Среда Delphi может создавать резервные копии главного файла проекта и файлов модулей и описаний форм. Резервные копии файлов содержат в расширении знак «~» (тильда) в качестве первого символа и создаются при повторном сохранении проекта для тех файлов, в исходном коде которых были сделаны изменения:

`*.~DP` —резервная копия главного файла проекта;

`*.~PA` — резервная копия модуля;

`*.~DF` — резервная копия файла описания формы.

Задать режим сохранения резервных копий можно с помощью диалогового окна команды меню **Tools → Editor Options**, установив на странице **Display** опцию **Create backup file** (рис. 4.12).

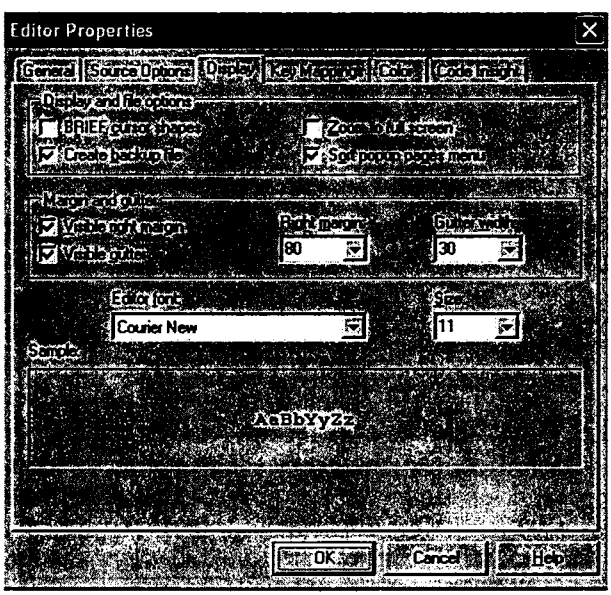


Рис. 4.12. Установка параметров редактирования

4.2.3. Компиляция и выполнение проекта

Компиляция проекта

Компиляция может быть выполнена на любой стадии разработки проекта и позволяет проверить внешний вид интерфейсных окон и правильность функционирования фрагментов создаваемого кода.

Запуск процесса компиляции выполняется по команде меню **Project → Compile <Project_Name>**. В строке меню команды указано имя проекта, разработка которого выполняется в текущий момент. При сохранении проекта под другим именем, соответственно, должно измениться имя проекта в меню.

При компиляции происходит следующее:

- компиляция файлов всех модулей, содержимое которых изменилось после последней компиляции, и модулей, использующих их с помощью директивы `uses` (в результате создаются файлы с расширением `*.dcu`);
- создание исполняемого файла — приложения: в процессе компиляции проекта создается готовый к выполнению файл (`*.exe`) или динамически загружаемая библиотека (`*.dll`).

Само приложение является автономным и не требует при своей работе дополнительных файлов Delphi.

Имя приложения совпадает с именем файла проекта.

Ход процесса компиляции (рис. 4.13) будет отображаться на экране, если установить опцию **Show compiler progress** в меню **Tools→Environment Options→Preferences**.

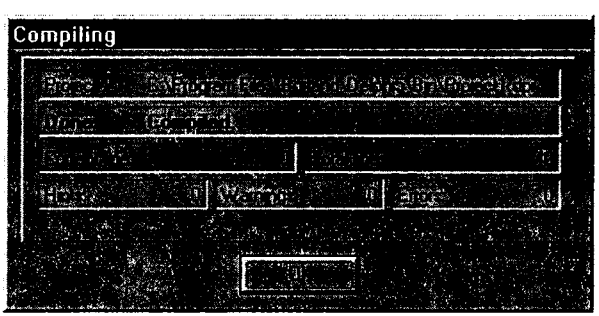


Рис. 4.13. Отображение процесса компиляции

Сборка проекта

Сборка проекта выполняется командой **Project→Build <Project_Name>**. При сборке перекомпилируются все модули, входящие в проект независимо от того, были в них внесены изменения или нет.

Запуск проекта

Запускать проект можно из среды Delphi и из среды Windows.

Запуск проекта из среды Delphi выполняется командой Главного меню **Run→Run**. Созданное приложение начинает свою работу. Если в модули программы были внесены изменения, то перед запуском проекта выполняется компиляция проекта.

При запуске проекта в среде Delphi необходимо учитывать следующие особенности:

- нельзя запустить вторую копию приложения;
- вносить изменения в модули (т. е. продолжить разработку проекта) можно только после завершения работы приложения;
- если приложение не может нормально завершить работу, необходимо выполнить завершение средствами Delphi с помощью команды меню **Run → Program Reset**.

Запуск проекта из среды Windows осуществляется так же, как и запуск любого другого приложения.

Отладка приложения

Для отладки приложений в среде Delphi можно использовать встроенную систему отладки. Проект в этом случае должен быть откомпилирован с отладочной информацией. Подключение отладчика происходит через установку опции **Debug Information** в окне параметров проекта (см. рис. 4.10).

Встроенный отладчик (Debugger) облегчает поиск и устранение ошибок в приложениях. Средства отладчика доступны с помощью команд пункта Главного меню **Run** и команды **View → Debug Windows**.

Система отладки предусматривает следующие действия:

- выполнение до указанного оператора (строки кода);
- пошаговое выполнение приложения;
- включение и выключение точек останова;
- выполнение приложения до точки останова;

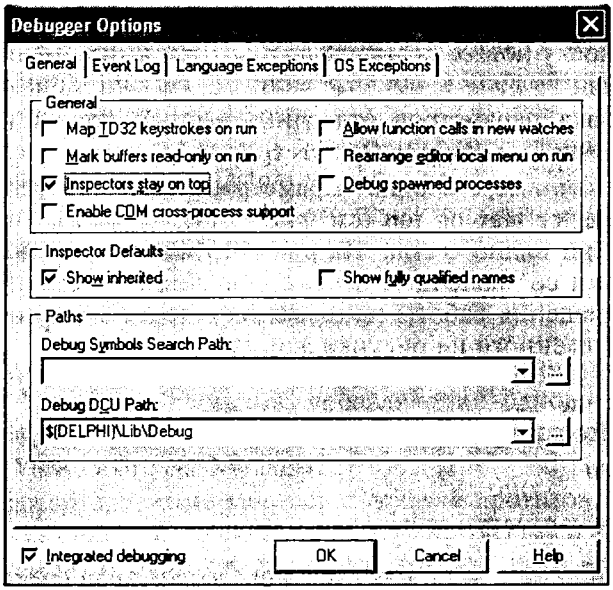


Рис. 4.14. Окно параметров отладчика

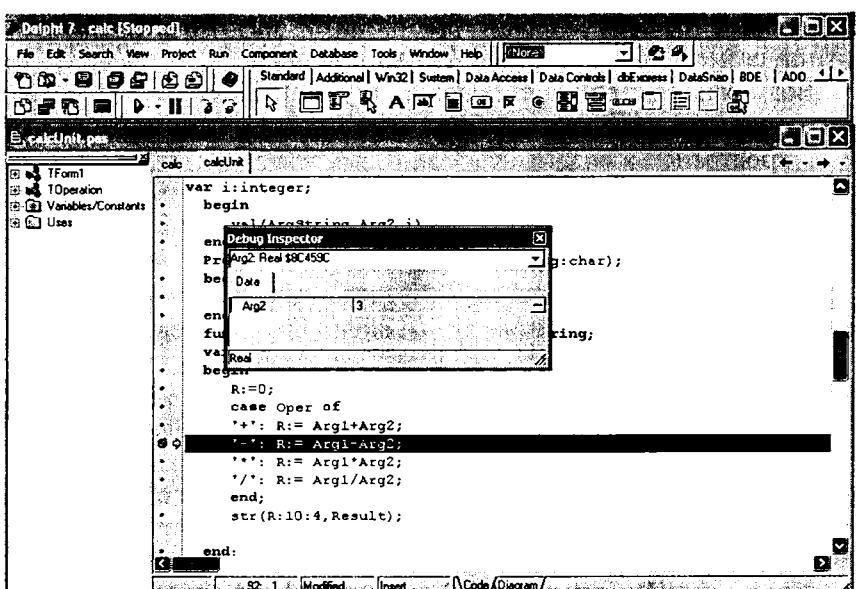


Рис. 4.15. Просмотр значения переменной в окне отладки

- просмотр значений данных в окнах просмотра;
- установку новых значений данных при выполнении приложения.

Установка параметров отладчика выполняется с помощью команды **Tools → Debugger Options** (рис. 4.14).

Если, например, в окне параметров отладчика установлена опция **Inspector stay on top** (см. рис. 4.14), то окно Инспектора отладки будет видно всегда. На рис. 4.15 показано окно инспектора отладки со значением переменной `Arg2` в момент временного прекращения работы приложения в точке останова (строке текста, выделенного на рисунке жирной точкой).

4.2.4. Средства управления параметрами проекта и среды разработки

Текущие параметры среды

Отображение и установка текущих параметров среды выполняются в диалоговом окне **Environment Options** (см. рис. 4.11), которое вызывается посредством команды меню **Tools→ Environment Options**. Параметры разбиты на группы, каждая группа размещена на отдельной странице окна.

Возможна настройка следующих групп параметров:

- страница **Preferences** — параметры конфигурации рабочего пространства среды;
- страница **Designer** — параметры Дизайнера Форм проекта;
- страница **Environment Variables** — просмотр и установка системных переменных, создание, редактирование и удаление пользовательских переопределений;
- страница **Object Inspector** — спецификация свойств Инспектора Объектов;
- страница **Library** — спецификация директорий размещения и параметров компиляции и компоновки библиотек проекта;
- страница **Palette** — параметры настройки содержимого страниц Палитры компонентов;
- страница **Explorer** — параметры настройки обозревателя проекта (**Project Browser**) и навигатора проекта (**Code Explorer**).

- страница **Type Library** — параметры настройки редактора Библиотеки типов (набора информации о типах объектов);
- страница **Internet** — типы файлов и параметры скриптов для приложений WebSnap;
- страница **Delphi Direct** — параметры доступа по сети к последним новостям Delphi на сайте `borland.com`;

Параметры среды для каждого проекта сохраняются в файле конфигурации с расширением `.cfg`.

Менеджер проекта (Project Manager)

В состав интегрированной среды разработки приложений включен Менеджер проектов, управляющий одновременно несколькими проектами. Окно Менеджера открывается по команде меню **View → Project Manager.**

Менеджер проектов работает с *группой* проектов, в которой может быть один или несколько проектов. Новый проект может быть добавлен в группу с помощью контекстного меню, когда выделена строка **ProjectGroup1**.

Менеджер проектов отображает иерархическую структуру группы проектов, самих проектов, а также все формы и модули, входящие в каждый проект. Для работы с проектами в группе можно использовать панель инструментов или контекстное

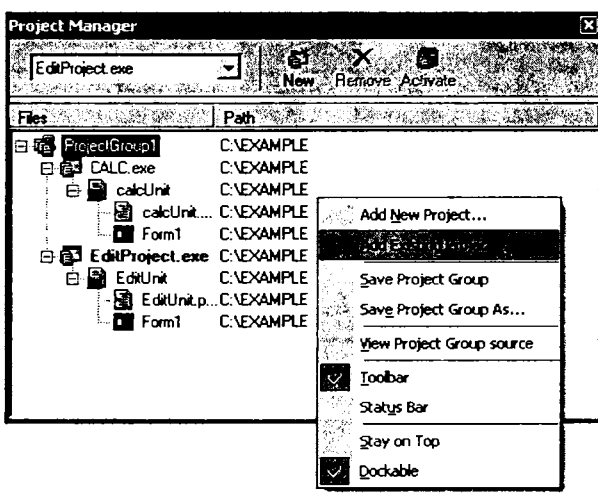


Рис. 4.16. Окно Менеджера проектов

меню, набор команд которого зависит от того, какой объект выделен в группе. На рис. 4.16 показано окно Менеджера проектов с контекстным меню управления приложением, позволяющим:

- добавить модуль в проект;
- удалить модуль из проекта;
- сохранить проект;
- изменить параметры проекта;
- откомпилировать проект и построить приложение и т. п.

Менеджер Проекта позволяет объединять проекты, которые работают вместе, в одну проектную группу. Группы проектов сохраняются в файлах с расширением `.bpg` (Borland Project Group).

Обозреватель проекта (Project Browser)

С помощью обозревателя проекта разработчик приложения может просматривать перечень модулей, классов, типов, свойств, методов, переменных и процедур, объявленных в проекте или используемых им (рис. 4.17). Окно Обозревателя проекта открывается по команде меню **View → Browser**. Перед использованием Обозревателя проект должен быть сохранен и откомпилирован.

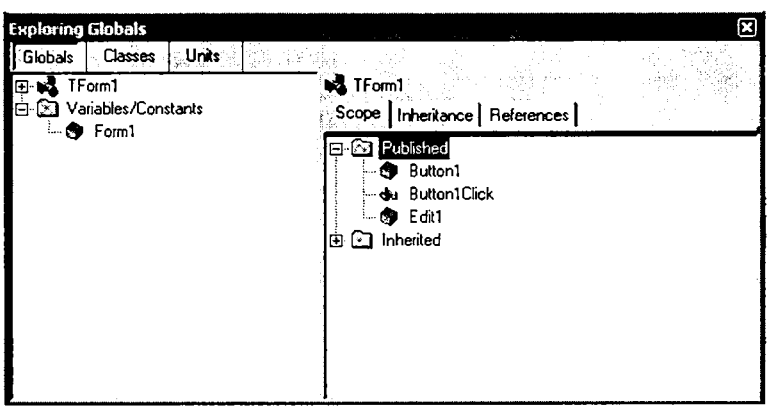


Рис. 4.17. Окно обозревателя проекта

Окно Обозревателя проекта разделено на две части. Слева в иерархическом виде отображаются доступные объекты, а справа для выбранного объекта более детально отображаются его характеристики.

Для просмотра в левой части окна доступны иерархии трех типов объектов, расположенные на различных страницах:

- **Globals** (Глобальные объявления);
- **Classes** (Классы объектов);
- **Units** (Модули).

В правой части окна доступны для просмотра следующие характеристики:

- **Scope** — список идентификаторов, объявленных в классе или модуле, выделенном в левой части окна;
- **Inheritance** — иерархическое дерево выделенного в левой части класса;
- **References** — имена файлов и номера строк программного кода текущего проекта, где есть ссылка на выделенный в левой части объект.

Для управления параметрами отображения объектов в Обозревателе используется команда меню **Tools→Environment Options→Explorer**.

С помощью Обозревателя проекта можно перемещаться по списку используемых проектом модулей и просматривать объекты в разделах `interface` или `implementation`, а также просматривать глобальные объявления и переходить к ссылкам на них в исходном коде.

Репозиторий объектов

Репозиторий (хранилище) объектов используется для создания новых элементов любого типа: форм, модулей данных, библиотек, компонентов и др. Объекты Репозитория рассматриваются в качестве шаблонов при разработке приложений.

Окно Репозитория открывается по команде меню **File→New→Other...** . Объекты-шаблоны сгруппированы. Каждая группа объектов размещена на отдельной странице:

- **New** — базовые объекты;
- **ActiveX** — объекты ActiveX и OLE;
- **Multitier** — объекты многопоточного приложения;
- **Projects** — проекты;
- **Form** — формы;
- **Dialogs** — диалоги;
- **Data Modules** — модули данных;
- **Business** — мастера форм.

Страница с названием **Example** (рис. 4.18) содержит форму текущего проекта. При добавлении или удалении формы проекта ее шаблон соответственно добавляется или исключается из Репозитория объектов.

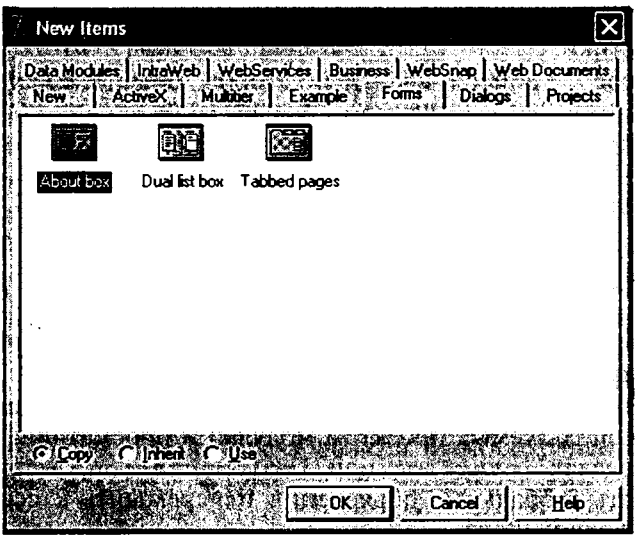


Рис. 4.18. Окно Репозитория объектов

Разработанный и отлаженный проект тоже может быть помещен в Репозиторий. Для этого необходимо инициировать команду меню **Project→Add to Repository** и заполнить появившееся диалоговое окно. Точно так же, как и шаблоны проектов, можно добавлять в Репозиторий и шаблоны форм, т. е. выбрать форму, которую предстоит добавить в Репозиторий, и инициировать команду контекстного меню **Add to Repository**.

Добавить в текущий проект объект из Репозитория можно одним из трех способов:

- **Copy** — в проект добавляется копия выделенного объекта Репозитория. Все изменения в объекте являются локальными и не затрагивают оригинал;
- **Inherit** — в проект добавляется новый объект, который наследует свойства выделенного объекта Репозитория. При каждой новой компиляции проекта все изменения, внесенные в шаблон Репозитория, будут отражены в проекте.

- **Use** — объект Репозитория становится частью проекта. Изменения, вносимые в объект в рамках разработки проекта, на самом деле вносятся непосредственно в шаблон Репозитория.

Справочная система

В состав справочной системы Delphi входят:
- стандартная система справки;
- справочная помощь через Internet;
- контекстно-зависимая справочная помощь.

Окно стандартной системы справки вызывается с помощью команд меню **Help→Delphi Help**, **Help→Delphi Tools**, **Help→Windows SDK** и состоит из трех страниц — **Содержание**, **Предметный указатель** и **Поиск** (рис. 4.19).

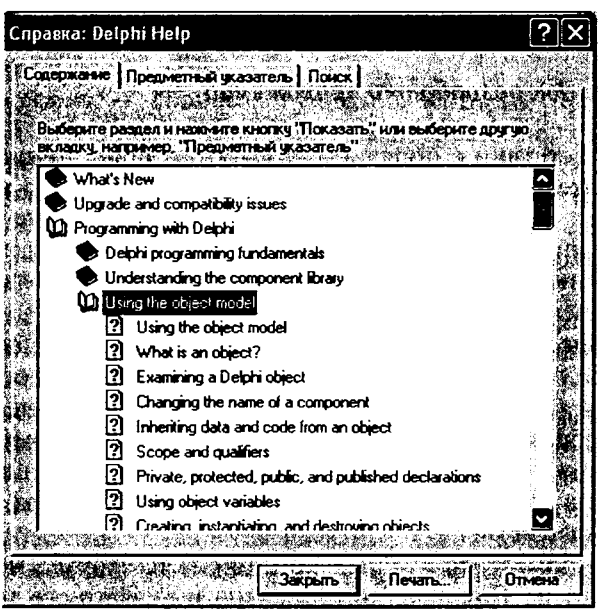


Рис. 4.19. Окно справки

Информация на странице **Содержание** представлена иерархией тематических разделов и подразделов.

Страница **Предметный указатель** обеспечивает доступ к справочной информации через словарь ключевых терминов.

Страница **Поиск** предназначена для проведения полнотекстового поиска в справочном массиве и отображения всех разделов справочной системы, в которых встречается указанная фраза или слово.

Команды меню, поддерживающие справочную помощь через Internet (например, **Help → Borland Home Page**, **Help → Delphi Home Page**), обеспечивают доступ на соответствующий сайт средствами браузера Internet.

Контекстно-зависимая справочная помощь вызывается путем нажатия клавиши <**F1**> функциональной клавиатуры. Содержимое справочной информации зависит от текущего состояния среды и связано с активизированным в текущий момент объектом.

4.3. Разработка приложения в среде Delphi

4.3.1. Порядок разработки приложений

В соответствии с порядком разработки приложений с графическим интерфейсом (см. гл. 2, п. 2.2.1) необходимы последовательно два взаимосвязанных этапа: проектирование функционального интерфейса приложения и программирование процедур обработки событий, возникающих при работе пользователя с приложением.

В процессе *создания интерфейса приложения* (первый этап разработки приложения) происходит последовательное размещение в окне Формы (по умолчанию — **Form1**) компонентов, наилучшим образом соответствующих функциональному назначению приложения. Для установки параметров размещения компонента на форме используется окно Инспектора Объектов (**Object Inspector**).

Чтобы поместить нужный компонент на форму, необходимо выбрать его из Палитры компонентов и указать его местоположение на области формы. После размещения компонента на форме можно изменять с помощью манипулятора «мышь» его положение и размеры.

По умолчанию компоненты выравниваются на форме по линиям сетки, которая при проектировании отображается на поверхности формы.

Выделение нескольких компонентов на форме выполняется с помощью мыши при нажатой клавише <**Shift**>.

Редактировать размещение компонентов можно с помощью контекстного меню или группы команд меню **Edit**, например:

- **Align** — выравнивание группы компонентов;
- **Bring to front** — перевод компонента на передний план;
- **Send to Back** — перевод компонента на задний план;
- **Size** — установка новых размеров компонента.

Следующий шаг — *определение свойств компонентов при проектировании*.

По типам хранящихся в них данных свойства делятся на следующие группы:

- *простые* — свойства, значения которых являются числами или строками (например, `Caption`, `Name`, `Left`, `Top`);
- *перечисляемые* — свойства, которые могут принимать значения из предложенного набора (списка) (например, свойства `Visible` и `Enabled` могут принимать значение `True` или `False`);
- *множества* — свойства, значения которых представляют собой комбинацию значений из предлагаемого множества (например, свойство формы `BorderIcon`);
- *объекты* — свойства, значения которых представляют собой объекты. В области значения свойства-объекта в скобках указывается тип объекта (например, свойство `Font` типа `TFont`).

Управлять свойствами компонента можно непосредственно в окне Конструктора формы или с помощью Инспектора объектов.

Для доступа к свойствам компонента через окно Инспектора объектов необходимо сначала в раскрывающемся списке верхней части окна, где отображаются название компонента и его тип, выбрать нужный компонент. В *левой* части окна Инспектора объектов приводятся названия всех свойств выбранного компонента, которые доступны на этапе разработки приложения. Для каждого свойства *справа* содержится значение этого свойства.

Чтобы изменить значения свойств, необходимо выбрать изменяемое свойство и, в зависимости от типа данных свойства выполнить следующие действия:

- в случае простого свойства — ввести в правой колонке Инспектора объектов новое значение;

- в случае перечисляемого свойства — открыть список, появившийся в соответствующей ячейке правой колонки, и выбрать нужное значение;
- в случае множества — установить значение True на выбираемых элементах предлагаемого множества;
- в случае объекта — заполнить отдельно значение каждого поля объекта или воспользоваться вспомогательными диалоговыми окнами задания значений.

Утверждается изменение свойства нажатием клавиши <**Enter**>, отменяется — нажатием клавиши <**Esc**>. Если для свойства введено неправильное значение, то выдается предупреждающее сообщение. Результат изменения свойств компонента сразу отображается в окне Конструктора формы.

На рис. 4.20 показано изменение простого свойства формы `Caption` (введено значение «Новая форма»), перечисляемого свойства `Color` и свойства типа множество — `BorderIcon`. На рис. 4.21 представлено окно заполнения свойства типа объект — установка шрифта (свойство `Font`).

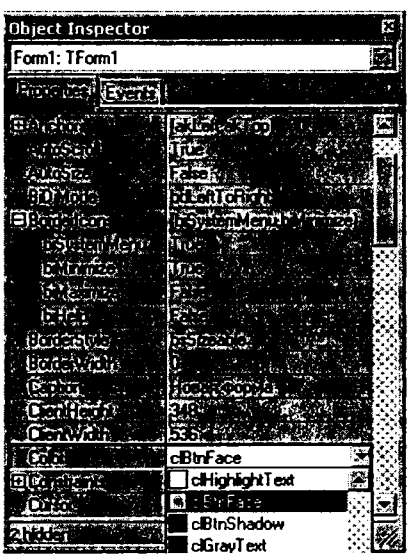


Рис. 4.20. Изменение свойств формы

При выполнении приложения свойства компонентов можно изменять с помощью операторов присваивания. Например, из-

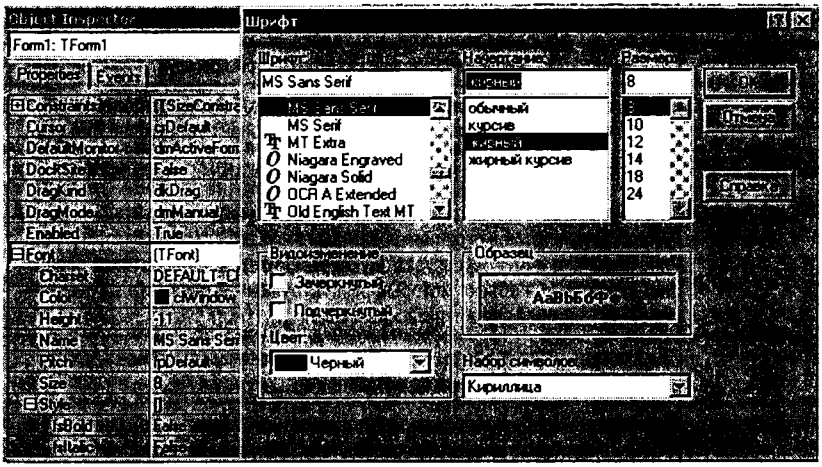


Рис. 4.21. Изменение свойства Font

менить заголовок формы можно программно путем оператора присваивания:

```
Form1.Caption:= 'Новая форма';
```

Функциональная схема работы приложения определяется процедурами, которые выполняются при возникновении определенных событий, происходящих при взаимодействии пользователя с управляющими элементами формы. Реакция на события присуща каждой форме и не зависит от назначения приложения и его особенностей.

На втором этапе разработки приложения для каждого компонента, размещенного на форме, разработчик должен определить нужную реакцию на те или иные действия пользователя (например, нажатие кнопки или выбор переключателя).

Каждый компонент имеет свой набор событий, на которые он может реагировать. Вместе с тем, существуют две категории стандартных событий, определенных для всех визуальных компонентов.

К первой категории относятся *события*, определенные *для всех без исключения визуальных компонентов*, а ко второй — *события*, характерные *только для оконных визуальных компонентов*. В табл. 4.16 представлены события, общие для всех визуальных компонентов, а в табл. 4.17 — события оконных визуальных компонентов.

Таблица 4.16. **События, общие для всех визуальных компонентов**

Событие	Момент возникновения и параметры процедур
`OnClick`	Возникает при нажатии левой клавиши мыши в области компонента или при нажатии клавиши <Enter>, если на компонент установлен фокус
`OnDblClick`	Возникает при двойном нажатии левой клавиши мыши в области компонента
`OnDragDrop`	Возникает в случае, когда пользователь «отпустил» перетаскиваемый объект. Параметр `Source` соответствующей процедуры-обработчика содержит указатель на объект, который был «отпущен», а параметр `Sender` — на объект, принявший «перетаскиваемый»
`OnDragOver`	Возникает в случае, когда пользователь «перетаскивает» объект для его размещения в рамках другого компонента.
`OnEndDrag`	Возникает в конце процедуры «перетаскивания» объекта. Параметр `Sender` соответствующей процедуры-обработчика указывает на «перетаскиваемый» объект, а параметр `Target` — на его образ в рамках принимающего компонента
`OnMouseDown`	Возникает при нажатии любой клавиши мыши в области компонента. Параметр `Button` соответствующей процедуры-обработчика этого события позволяет определить, какая клавиша была нажата, параметр `Shift` — были ли нажаты клавиши <Shift>, <Ctrl> или <Alt> вместе с клавишей мыши, а параметры `X` и `Y` содержат координаты курсора мыши в момент нажатия клавиши
`OnMouseMove`	Возникает при «буксировке» манипулятора «мышь» (т. е. при его перемещении при нажатой левой клавише). Параметр `Shift` соответствующей процедуры-обработчика определяет, были ли нажаты клавиши <Shift>, <Ctrl> или <Alt> вместе с клавишей мыши, а параметры `X` и `Y` содержат координаты курсора мыши в момент нажатия клавиши
`OnMouseUp`	Парное для `OnMouseDown`. Возникает, когда ранее нажатая клавиша мыши отпущена. Имеет те же параметры, что и `OnMouseDown`

Процедура, связанная с несколькими событиями для различных компонентов, называется *общим обработчиком* и вызывается при возникновении любого из связанных с ней событий.

Для создания процедуры обработки события нужно:

- выделить на форме компонент;
- перейти на страницу событий Инспектора Объектов;
- выделить событие, для которого будет создаваться процедура-обработчик события;
- посредством двойного нажатия мыши в области значения события получить доступ в модуль формы, где Delphi автоматически создаст заготовку процедуры-обработчика;

Таблица 4.17. **События оконных визуальных компонентов**

Событие	Момент возникновения и параметры процедуры
`OnEnter`	Возникает, когда компонент получает фокус ввода
`OnExit`	Возникает при потере фокуса компонентом
`OnKeyDown`	Возникает при нажатии любой клавиши на клавиатуре. Параметр `Key` соответствующей процедуры-обработчика имеет тип `Word` и может содержать коды виртуальных клавиш. Параметр `Shift` передает информацию о нажатии клавиш `<Shift>`, `<Ctrl>`, `<Alt>` и о нажатии клавиш мыши
`OnKeyPress`	Возникает при нажатии клавиши на клавиатуре. Параметр `Key` соответствующей процедуры-обработчика имеет тип `Char` и содержит ASCII-код нажатой клавиши. Для клавиш, которые не имеют ASCII-кодов (например, `<Shift>`, `<Ctrl>` или `<Alt>`), событие не возникает
`OnKeyUp`	Парное для `OnKeyDown`. Возникает, когда пользователь отпускает нажатую ранее клавишу. Имеет те же параметры, что и `OnKeyDown`

- в месте, где будет установлен текстовый курсор, написать код, который должен выполняться при возникновении события.

Итак, для обеспечения *выполнения функций* приложения необходимо:

- задать в Инспекторе Объектов значения свойств и процедур обработки событий;
- написать программный код для заданных процедур обработки событий.

4.3.2. Разработка приложения «Редактор текстов»

На примере разработки приложения «Редактор текстов» рассмотрим использование на форме компонентов различных типов, а также порядок проектирования главного меню, всплывающего меню и панели инструментов.

Разрабатываемое приложение должно обеспечить:

- отображение существующего текстового файла в окне редактирования или создание нового файла;
- выполнение простейших операций по редактированию: вставка-удаление символов, копирование и перемещение фрагментов текста с использованием Системного буфера обмена (ClipBoard);

- сохранение результатов редактирования в том же или в новом файле.

Рассмотрим последовательно этапы разработки приложения.

Проектирование интерфейса

Первый этап — проектирование интерфейса. Главное интерфейсное окно приложения будет содержать следующие визуальные компоненты:

- Главное меню, расположенное в верхней части окна;
- Панель инструментов, расположенную сразу под строкой Главного меню;
- область для отображения и редактирования текстов, которая должна занимать всю оставшуюся часть главного интерфейсного окна;
- «Всплывающее» меню, которое должно появляться при нажатии на правую клавишу манипулятора «мышь» в области редактирования.

Главное меню. Для организации выполнения перечисленных действий построим Главное меню приложения.

Разобьем функции приложения на две группы — **Файл** и **Редактировать**. Эти две группы и представят собой опции Главного меню.

В группу **Файл** войдут функции **Создать**, **Открыть**, **Сохранить**, **Сохранить как...** и **Выход**. Опция меню **Файл** должна объединять соответствующие пункты меню, представляющие собой команды, вызывающие выполнение перечисленных функций.

В группу **Редактировать** — функции **Вырезать**, **Копировать** и **Вставить** (функции работы с фрагментами текста через посредство Системного буфера обмена).

Дизайнер меню. Компонент Главного меню располагается на странице **Standard** (Стандартные) Палитры компонентов. Установим компонент на форме и вызовем Дизайнер меню. Вызвать Дизайнер меню можно либо через свойство `Items` компонента Главного меню, либо с помощью контекстного меню, которое раскрывается по нажатию на правую клавишу манипулятора «мышь», когда компонент Главного меню выделен на форме.

Каждый пункт меню вводится с помощью Дизайнера, а затем отображается в строке меню на форме. При вводе нового пункта на самом деле создается новый компонент и добавляется в список компонентов Инспектора объектов (хотя визуально на

форму ничего не добавляется). В свойствах компонента обязательно заполняются поля `Caption` — название пункта, `Name` — имя компонента и, если необходимо, поля `ShortCut` – «горячая клавиша» для вызова функции с помощью клавиатуры и `Bitmap` — для добавления пиктограммы к заголовку пункта. На рис. 4.22 показаны свойства компонента пункта меню «Сохранить», доступ к которым открыт через Дизайнер меню.

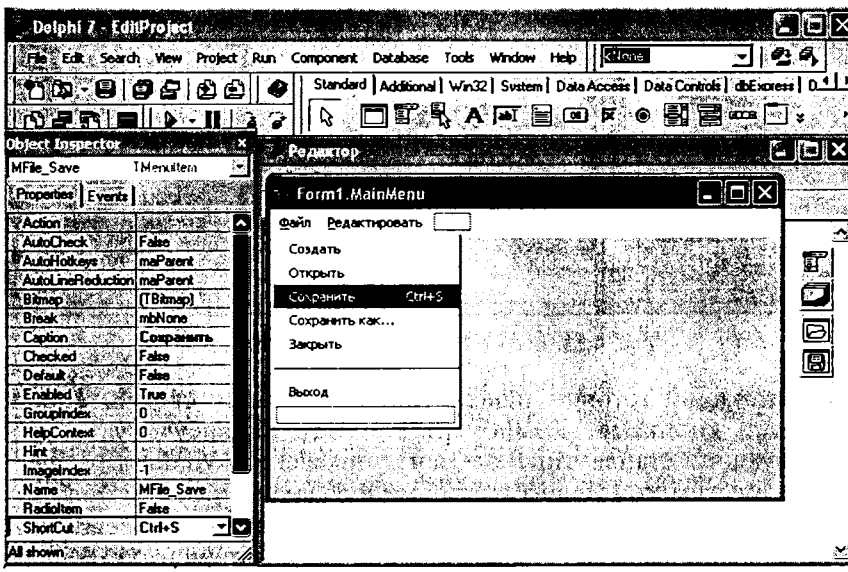


Рис. 4.22. Дизайнер меню

Для удобства восприятия команды меню визуально разбиваются на группы с помощью добавления в меню специальных пунктов-разделителей. Чтобы ввести в меню такой пункт, достаточно в свойстве `Caption` установить знак «-» (дефис), как показано на рис. 4.23.

Панель инструментов. Следующий этап проектирования формы — размещение Панели инструментов. Панель инструментов предназначена для дублирования наиболее часто использующихся команд меню графическими кнопками, что обеспечивает более быстрый альтернативный способ вызова связанных с командами меню функций.

Начиная с версии 3, в Delphi появился специальный компонент `ToolBar` (страница «Win32» Палитры компонентов), пре-

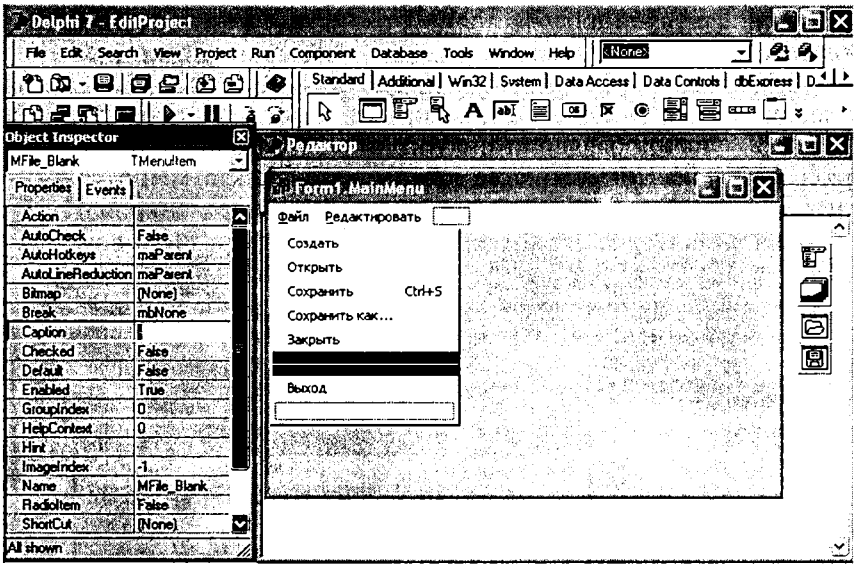


Рис. 4.23. Ввод в меню пункта-разделителя

доставляющий готовую панель инструментов со своими собственными кнопками. Компонент `ToolBar` инкапсулирует соответствующий стандартный элемент управления Windows.

Разместим компонент на форме и зададим значения некоторым свойствам (помимо свойства `Name`). Значение `alTop` свойства `Align` определяет положение панели инструментов на форме — в верхней части, сразу под строкой Главного меню. Установка свойства `Flat` в `true` задает особый стиль изображения кнопок панели — без прорисовки контура (контур появляется только у кнопки, на которую указывает курсор).

Сначала размещенная на форме панель инструментов пуста. Добавление кнопок осуществляется с помощью всплывающего контекстного меню компоненты: для добавления кнопки используется команда **New Button**, а для добавления разделителя — команда **New Separator**. Пример формы с компонентом `ToolBar` во время проектирования показан на рис. 4.24.

Компонент `ToolBar` содержит объекты класса `TToolButton`. Это *внутренние* объекты (аналогично объектам класса `TMenuItem`, которые являются внутренними для объекта `MainMenu`).

Далее, для каждой кнопки задается рисунок — пиктограмма, соответствующая функциональному назначению кнопки. Опре-

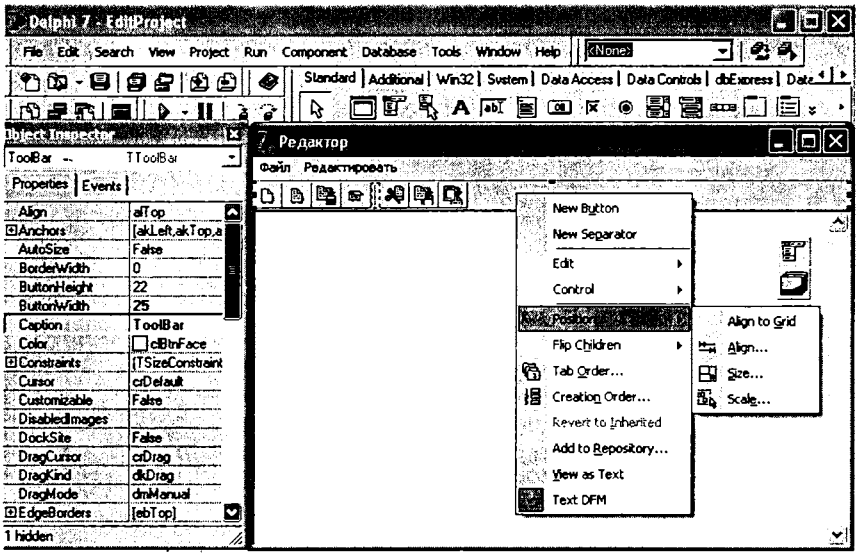


Рис. 4.24. Проектирование компонента **ToolBar**

деление рисунков для кнопок панели инструментов осуществляется с помощью *невизуального* компонента `ImageList`. Этот компонент не имеет своего места на форме и предназначен для создания коллекции рисунков, выбор которых происходит по индексу — номеру рисунка в коллекции. Имя компонента заносится в свойство `Images` Панели инструментов и, таким образом, обеспечивается связь каждой отдельной кнопки панели с соответствующей картинкой.

Коллекция рисунков ImageList. Для создания коллекции рисунков на форму добавляется компонента `ImageList` (страница **Win32** Палитры компонентов). Контекстное меню компонента позволяет вызвать Редактор **ImageList**, с помощью которого проводятся действия по добавлению и удалению рисунков в коллекции (рис. 4.25).

Каждый добавленный рисунок получает в коллекции свой индекс и может быть в дальнейшем связан с некоторой кнопкой панели инструментов путем занесения значения индекса в свойство `ImageIndex` кнопки.

На примере кнопки Панели инструментов продемонстрируем использование пары свойств `ShowHint` и `Hint` для формирования «всплывающих» подсказок. Установка свойства `ShowHint`

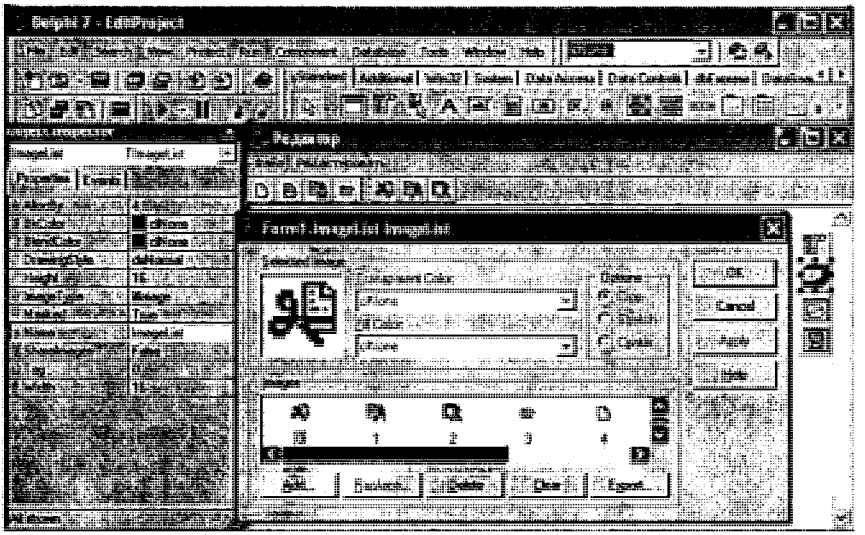


Рис. 4.25. Редактор ImageList — создание коллекции рисунков

в `True` обеспечивает включение аппарата «всплывающей» подсказки для компонента, т. е. появление в специфическом «всплывающем» окне текста, заданного в свойстве `Hint`, при задержке курсора в области компонента (рис. 4.26).

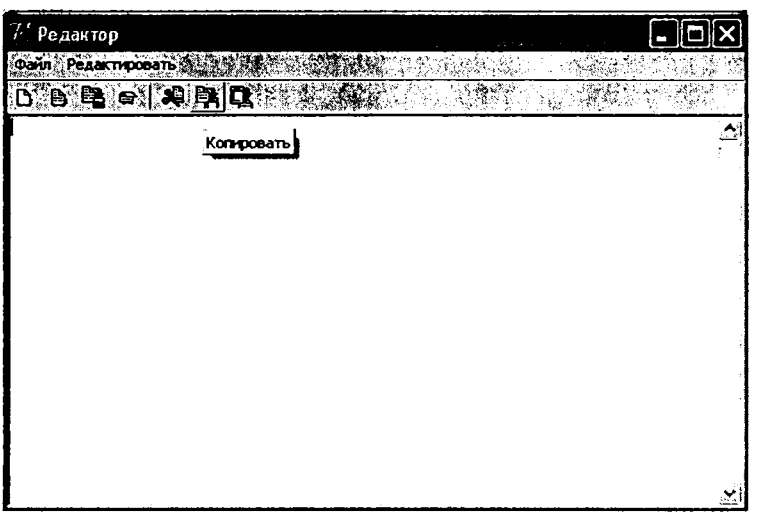


Рис. 4.26. «Всплывающая» подсказка

Область для отображения и редактирования текстов. Для отображения и редактирования текстовых файлов разместим на форме компонент `Memo` класса `TMemo`. Установим значение `Client` в свойстве `Align` компонента, обеспечив тем самым заполнение компонентом всей незанятой части главного интерфейсного окна.

Компонент `Memo` предназначен для ввода разделенных и не разделенных на строки текстов. При установке свойства `WordWrap` в `True` происходит дополнительное разбиение текстов на строки по ширине области компонента.

Свойство `ScrollBars` позволяет установить на области компонента стандартные полосы «прокрутки» текстов: `ssVertical` — только вертикальная полоса, `ccHorizontal` — только горизонтальная полоса, `ssBoth` — обе полосы «прокрутки» (в рассматриваемом примере установлена вертикальная полоса «прокрутки»).

Компонент автоматически поддерживает стандартные функции редактирования — ввод, замену и удаление отдельных символов, а также *выделение* фрагментов текста. Выделение фрагмента происходит либо с помощью «буксировки» манипулятора «мышь», либо посредством клавиатуры (с помощью клавиш перемещения курсора — «стрелка вправо», «стрелка влево» и т. п. — при нажатой клавише `<Shift>`).

Контекстное меню. Кроме Главного меню обычно еще используется похожий на него компонент `PopupMenu` (страница **Standard** Палитры компонентов). Такое меню отображается при нажатии на правую клавишу манипулятора в области компонента, свойство `PopupMenu` которого указывает на данное меню. Это стандартный способ использования контекстного (локального, всплывающего) меню.

Разместим на форме *неконный* компонент `PopupMenu` и добавим в меню опции из группы команд **Редактирование** Главного меню. Порядок проектирования контекстного меню полностью совпадает с порядком проектирования Главного меню и тоже предполагает использование Дизайнера меню.

Проектирование пунктов контекстного меню читателю предлагается провести самостоятельно.

Для привязки контекстного меню к компоненту `Memo` занесем в свойство `PopupMenu` компонента имя контекстного меню.

Использование стандартных диалогов. Для организации работы с файлами (открытия файлов для редактирования и сохране-

ния результатов) используются компоненты стандартных диалогов Windows, реализованные в Delphi. Разместим на форме приложения *неоконные* компоненты `OpenDialog` и `SaveDialog` (страница **Dialogs** — «Диалоги» — Палитры компонентов) и рассмотрим возможности их использования на примере компонента `OpenDialog` (`SaveDialog` работает похожим образом).

Использование диалоговых компонентов не обеспечивает автоматического открытия или сохранения файлов, но позволяет получить для дальнейшей обработки в программе *имя* выбранного пользователем файла (рис. 4.27).

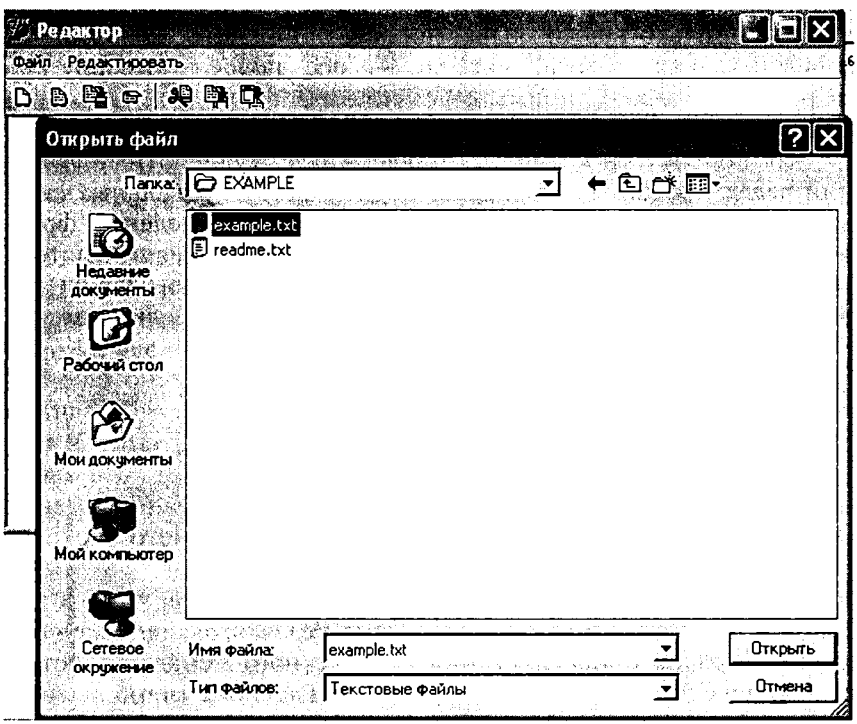


Рис. 4.27. Диалоговое окно «Открыть файл»

Рассмотрим некоторые свойства компонентов, позволяющие управлять видом и характером действия диалогового окна.

Свойство `FileName` назначает имя файла по умолчанию (имя, которое будет отображаться в окне **Имя файла** при откры-

тии диалога), а затем (после закрытия диалогового окна) содержит имя выбранного пользователем файла.

Свойство `DefaultExt` позволяет указать расширение имени файла, которое будет добавляться по умолчанию в том случае, если в окне **Имя файла** имя будет указано без расширения (в нашем примере используется значение `txt`).

Свойство `Filter` задает список файловых масок (файловых фильтров), из которого пользователь может выбирать для отображения в диалоговом окне только нужные типы файлов. Файловый фильтр представляет собой строку, состоящую из одной или более пар значений. Пара значений — это описание фильтра и сам фильтр (один или несколько, разделенных символом «;»). Каждая пара значений задает один фильтр. Пары значений и значения внутри одной пары разделяются символом «|» (вертикальная черта). Например, следующая строка задает два фильтра для диалога:

```
MyOpenDialog.Filter := 'Текстовые файлы (*.txt)|*.txt|
                       Все файлы (*.*)|*.*'
```

На рис. 4.28 показано окно для заполнения свойства `Filter` при проектировании формы.

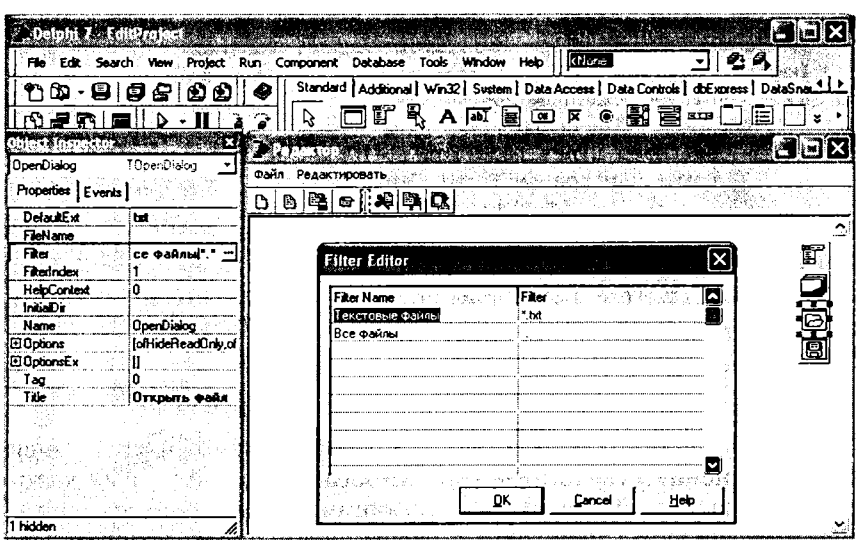


Рис. 4.28. Свойства компоненты OpenDialog (заполнение свойства Filter)

Разработка процедур обработки событий

Второй этап разработки приложения — разработка процедур обработки событий. Рассмотрим последовательно процедуры-обработчики для пунктов Главного меню (обработчики события `OnClick` для каждого компонента класса `TMenuItem`).

Пункт Главного меню **Файл → Открыть** предназначен для организации диалога выбора файла на редактирование. Метод `OpenDialog.Execute` открывает диалоговое окно, а после нажатия на кнопку «Открыть» в диалоговом окне возвращает значение `True` и заполняет свойство `OpenDialog.FileName`, помещая в него имя выбранного в окне диалога файла.

Метод `LoadFromFile` объекта `TStrings`, представляющего свойство `Lines` компонента `EditMemo`, позволяет одновременно организовать открытие и чтение файла в область редактирования. Если процесс загрузки файла закончился удачно, изменяется заголовок окна приложения (в него помещается имя открытого файла) и заполняется глобальная переменная `CurrentFileName`, служащая для хранения имени текущего файла. В случае возникновения ошибки при загрузке файла управление передается в блок обработки исключений и выдается диалоговое окно сообщения.

Текст процедуры:

```
procedure TForm1.MFile_OpenClick(Sender: TObject);
begin
if OpenDialog.Execute then
    try
       EditMemo.Lines.LoadFromFile(OpenDialog.FileName);
       Form1.Caption := Form1.Caption + ' - ' +
                   OpenDialog.FileName;
       CurrentFileName := OpenDialog.FileName;
    except
       on EFOpenError do
         MessageDlg('Ошибка при открытии файда', mtError,
                        [mbOk], 0);
    end;
end;
```

Пункт Главного меню **Файл → Сохранить как...** предназначен для сохранения результатов редактирования в файле, имя которому назначает пользователь. При организации процесса сохранения результатов необходимо учесть возможность существования файла с именем, указанным в окне диалога сохранения.

В предлагаемой процедуре-обработчике используется стандартная функция FileExists для проверки существования файла и организуется запрос-диалог для принятия решения. Стандартная функция MessageDlg открывает диалоговое окно с тремя управляющими кнопками — [Yes], [No] и [Cancel] и возвращает константу — признак выполненного действия (нажатой кнопки).

В зависимости от принятого пользователем решения выполняется одна из трех последовательностей действий: сохранение результатов редактирования с помощью применения метода SaveToFile, симметричного методу LoadFromFile; повторяется диалог назначения имени файла или происходит выход из процедуры-обработчика:

```
procedure TForm1.MFile_SaveAsClick(Sender: TObject);
label 1;
var
   btn : Word;
   fsave : Boolean;

begin
if EditMemo.Text <> '' then
begin
1: if SaveDialog.Execute then
   if FileExists(SaveDialog.FileName) then
    begin btn := MessageDlg('Файл существует. Перезаписать?',
                    mtWarning, [mbYes, mbNo, mbCancel], 0);
          if btn = mrNo then goto 1;
          if btn = mrCancel then fsave := false;
          if btn = mrYes then fsave := true;
    end
   else fsave := true;
   if fsave then
    try
      EditMemo.Lines.SaveToFile(SaveDialog.FileName);
      CurrentFileName := SaveDialog.FileName;
      Form1.Caption := ProgCaption + ' - ' +
                          SaveDialog.FileName;
    except
      on EInOutError do
       MessageDlg('Ошибка записи в файл', mtError, [mbOk], 0);
    end;
end;
```

Посредством пункта Главного меню **Файл → Сохранить** результаты редактирования сохраняются в текущем файле, если

имя текущего файла не пусто, или происходит вызов процедуры-обработчика для пункта меню **Файл → Сохранить как...** :

```
procedure TForm1.MFile_SaveClick(Sender: TObject);
label 1;
var
   btn : Word;
   fsave : Boolean;

begin
if EditMemo.Text <> '' then
 if CurrentFileName = '' then MFile_SaveAs.Click
 else
    try
      EditMemo.Lines.SaveToFile(CurrentFileName);
    except
      on EInOutError do
       MessageDlg('Ошибка записи в файл', mtError, [mbOk], 0);
    end;
end;
```

Обработчики событий пунктов Главного меню из группы **Редактировать** построены на использовании методов работы с Системным буфером обмена (Clipboard). Рассмотрим эти методы, общие для компонентов, поддерживающих процессы редактирования.

Метод `CopyToClipboard` предназначен для копирования *выделенного фрагмента* текста в Системный буфер обмена.

Метод `CutToClipboard` предназначен для удаления из текста *выделенного фрагмента* и переноса его в Системный буфер обмена.

Метод `PasteFromClipboard` предназначен для вставки текста, содержащегося в Системном буфере обмена, по месту текстового курсора.

Использование этих методов позволяет организовать процедуры редактирования, связанные не только с перемещением фрагментов текста внутри области редактирования, но и обеспечить взаимодействие разрабатываемого приложения с другими, поддерживающими обработку текстовой информации.

Процедуры-обработчики, реализованные для пунктов Главного меню, являются *общими* и подключаются к соответствующим кнопкам панели инструментов и соответствующим пунктам контекстного меню.

Окно работающего приложения представлено на рис. 4.29.

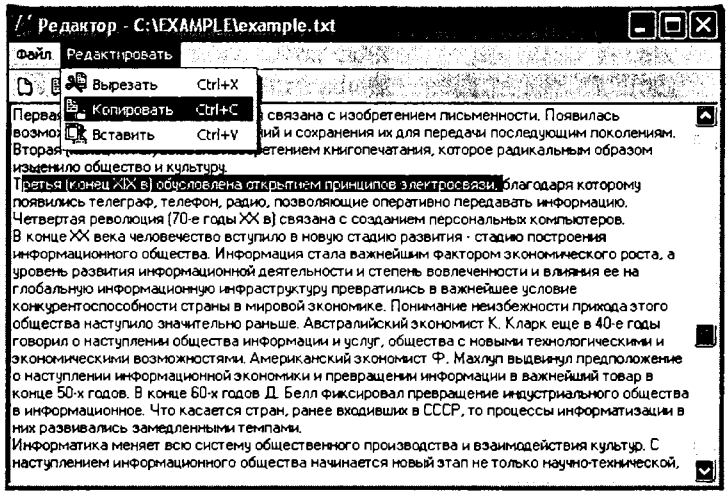


Рис. 4.29. Приложение «Редактор текстов»

4.4. Архитектура приложений, работающих с внешними источниками данных (базами данных)

Работа с внешними источниками данных подразумевает:

- получение данных;
- представление данных в определенном формате для просмотра пользователем;
- обработку (редактирование) в соответствии с реализованными в программе алгоритмами;
- возврат обработанных данных в источник данных.

Источник данных — это совокупность данных в определенном формате представления и программных средств, обеспечивающих поддержку формата, управление и манипулирование данными (например, базу данных и СУБД). В качестве источника данных могут выступать базы данных, текстовые файлы, электронные таблицы и т. п.

Несмотря на разнообразие программного обеспечения ведения источников данных, общая архитектура приложения, работающего с источниками данных, остается неизменной и включает следующие механизмы (рис. 4.30):

- механизм соединения с источником данных, обеспечивающий двунаправленный поток данных от программных

Пользовательский интерфейс. Содержит компоненты отображения данных и обеспечивает просмотр, редактирование и управление данными

Механизм связи внутреннего представления с интерфейсом приложения. Содержит компоненты, обеспечивающие передачу данных в визуальные компоненты и возврат результатов редактирования в набор данных

Механизм внутреннего представления данных. Содержит компоненты наборов данных, обеспечивающие хранение полученных данных в приложении и предоставление их по запросам. Общий предок всех компонентов — TDataSet

Механизм получения и отправки данных. Содержит компоненты, обеспечивающие соединение с источником данных и двунаправленный поток данных

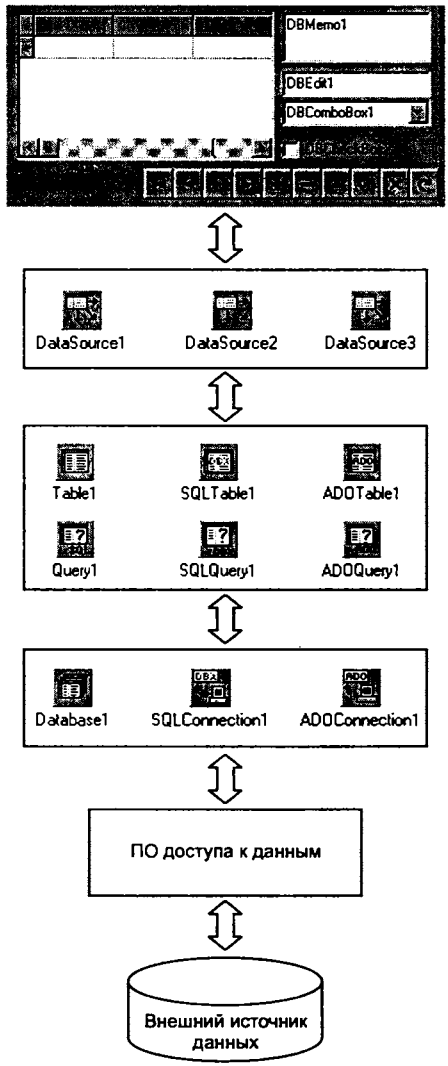


Рис. 4.30. Механизм доступа к внешнему источнику данных

средств ведения источника данных до приложения и обратно;

- механизм внутреннего представления данных, обеспечивающий хранение данных и реализацию запросов на их предоставление разным частям приложения;

- механизм связи внутреннего представления с элементами пользовательского интерфейса;
- пользовательский интерфейс, реализующий предоставление данных пользователю на обработку;
- алгоритмы обработки данных (бизнес-логику).

Между приложением и собственно источником данных работает специальное программное обеспечение (так называемое промежуточное ПО), управляющее процессом обмена данными. Промежуточное ПО может быть реализовано разными способами, например:

- как программное окружение приложения, без которого приложение не сможет работать;
- как набор драйверов и динамических библиотек;
- как подпрограммы, интегрированные в само приложение;
- как отдельный сервер, обслуживающий множество приложений.

Приложения Delphi могут осуществлять доступ к внешним источникам данных с использованием следующих технологий:

BDE (Borland Database Engine) — процессор баз данных фирмы Borland. BDE представляет собой совокупность динамических библиотек и драйверов, обеспечивающих доступ к данным. Процессор BDE должен устанавливаться на всех компьютерах, на которых выполняются Delphi-приложения, работающие с источниками данных. Приложение посредством BDE передает запрос к источнику данных, а обратно получает требуемые данные.

ADO (ActiveX Data Objects — объекты данных ActiveX) осуществляет доступ к информации с помощью OLE DB (Object Linking and Embedding Data Base — связывание и внедрение объектов баз данных). Механизм ADO является стандартом фирмы Microsoft. Использование этой технологии подразумевает использование настраиваемых провайдеров данных. Технология ADO основана на стандартных интерфейсах COM, являющихся системным механизмом Windows. Это позволяет удобно распространять приложения баз данных без вспомогательных библиотек.

dbExpress — технология, в соответствии с которой обеспечение взаимодействия с серверами баз данных основано на использовании специализированных драйверов. Последние для получения данных применяют запросы SQL. На стороне клиента при этом не обеспечивается возможность прямого редактирования наборов данных.

InterBase — технология, реализующая неопосредственный доступ к базам данных InterBase.

Итак, чтобы обеспечить в приложении работу с таблицей источника данных, необходимо:

1. Выбрать одну из предоставляемых средой технологий доступа к источнику данных.

2. Разместить на форме приложения компонент, устанавливающий соединение с источником данных, и настроить его на источник данных.

3. Разместить на форме приложения компонент — набор данных для хранения данных, получаемых из источника данных. Набор данных при этом может быть связан с источником данных таким образом, что все изменения, произведенные в наборе данных, будут фиксироваться в источнике данных. Выбор компонента определяется технологией доступа и потребностями приложения (таблица набора данных целиком или результат SQL-запроса).

4. Разместить на форме приложения компонент типа `TDataSource`, обеспечивающий связь набора данных с визуальными компонентами отображения данных.

5. Разместить на форме приложения визуальные компоненты отображения данных, обеспечивающие просмотр, редактирование и управление данными.

Таким образом, приложения могут получать доступ к источникам данных с помощью разнообразных технологий доступа, но любое приложение, работающее с источником данных в Delphi, имеет стандартный набор базовых компонентов, который является единой основой технологии доступа к данным. Это позволяет унифицировать процесс разработки приложений, использующих внешние источники данных.

4.4.1. Набор данных

Любое приложение, работающее с внешним источником данных, должно уметь выполнять как минимум две операции:

1) подключаться к источнику данных и считывать имеющуюся в таблицах информацию. Эта функция в значительной степени зависит от реализации конкретной технологии доступа к данным;

2) обеспечивать представление и редактирование полученных данных.

Множество записей одной или нескольких таблиц, переданные в приложение в результате активизации компонента доступа к данным, будем называть *набором данных*. Для представления какой-либо группы записей используются возможности класса, который инкапсулирует набор данных и содержит свойства и методы для управления записями и полями.

Класс `TDataSet` является базовым классом иерархии классов, поддерживающих различные технологии доступа к данным: он инкапсулирует абстрактный набор данных и максимально реализует общие методы работы с ним (рис. 4.31).

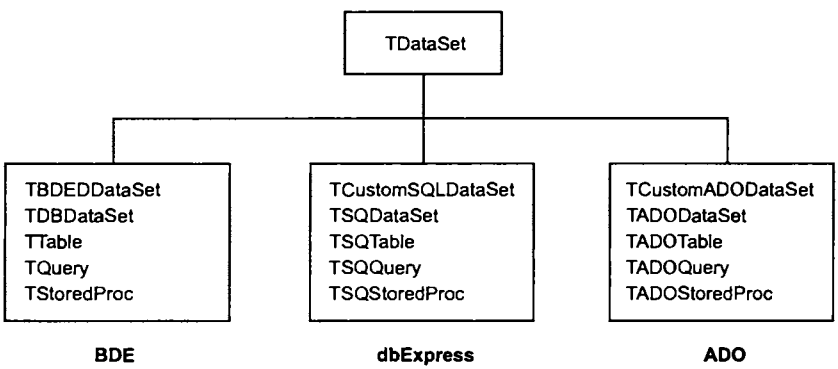


Рис. 4.31. Иерархия классов, обеспечивающих функционирование набора данных

На основе базового класса реализованы специальные компоненты для различных технологий доступа к данным, которые позволяют разработчику строить приложения, используя одни и те же приемы и настраивая одинаковые свойства.

В основе иерархии классов, обеспечивающих функционирование наборов данных в приложениях Delphi, лежит класс `TDataSet`. Этот класс задает структурную основу функционирования набора данных: к его методам необходимо лишь добавить требуемые вызовы соответствующих функций реальных технологий. Кроме того, класс `TDataSet` может использоваться разработчиками в качестве основы для создания собственных компонентов. Основные механизмы, реализованные в классе `TDataSet`, представлены в табл. 4.18.

Таблица 4.18. **Некоторые свойства и методы класса TDataSet**

Свойство/метод	Описание
`property Active: Boolean;`	Установка активного (`True`) и неактивного (`False`) состояния набора данных
`procedure Open;`	Открывает набор данных — переводит в активное состояние
`procedure Close;`	Закрывает набор данных — переводит в неактивное состояние
`procedure Next;`	Перемещение курсора на одну запись вперед (вниз)
`procedure Prior;`	Перемещение курсора на одну запись назад (вверх)
`procedure First;`	Перемещение курсора на первую запись
`procedure Last;`	Перемещение курсора на последнюю запись
`property Eof: Boolean;`	Признак, указывающий, достигнута (`True`) или нет (`False`) последняя запись набора
`property Bof: Boolean;`	Признак, указывающий, достигнута (`True`) или нет (`False`) первая запись набора
`function MoveBy(Distance: Integer): Integer;`	Перемещение вперед и назад на заданное число записей. Параметр `Distance` определяет число записей. Если параметр отрицательный — перемещение осуществляется к началу набора данных, иначе — к концу
`procedure DisableControls;`	Отключение всех связанных компонентов отображения данных (используется для ускорения перемещения по набору данных)
`procedure EnableControls;`	Операция, обратная `DisableControls`
`property RecordCount: Integer;`	Общее число записей набора данных.
`function IsEmpty: Boolean;`	Возвращает значение `True`, если набор данных пуст
`property RecNo: Integer;`	Номер текущей записи
`property RecordSize: Word;`	Размер записи в байтах
`property Fields: TFields;`	Инкапсулирует совокупность полей набора данных
`property FieldDefs: TFieldDefs;`	Параметры полей
`property FieldCount: Integer;`	Общее число полей набора данных

Окончание табл. 4.18

Свойство/метод	Описание
`property FieldValues[const FieldName: string]: Variant; default;`	Доступ к значениям полей текущей записи. Параметр `FieldName` задает имя поля
`function FieldByName(const FieldName: string): TField;`	Доступ к параметрам поля. Параметр `FieldName` задает имя поля
`procedure GetFieldNames (List: TStrings);`	Возвращает в параметре `List` полный список имен полей набора данных
`property CanModify: Boolean;`	Принимает значение `True` для редактируемых наборов
`procedure Edit;`	Переводит набор данных в режим редактирования
`procedure Post; virtual;`	Сохранение изменений (вызывается автоматически самим набором данных при переходе на другую запись)
`procedure Cancel; virtual;`	Отменяет все изменения, сделанные после последнего вызова метода `Post`
`procedure Append;`	В конец набора данных добавляется новая пустая запись (набор данных переходит в режим редактирования самостоятельно)
`procedure Insert;`	Новая пустая запись добавляется на место текущей, а текущая запись и все нижеследующие смещаются на одну позицию вниз (набор данных переходит в режим редактирования самостоятельно)
`procedure AppendRecord(const Values: array of const);`	Добавление в конец новой записи со значениями полей
`procedure InsertRecord(const Values: array of const);`	Вставка новой записи со значениями полей
`procedure SetFields(const Values: array of const);`	Заполнение полей существующей записи
`procedure Delete;`	Удаление текущей записи
`procedure ClearFields;`	Очистить содержимое всех полей текущей записи. Поля при этом принимают значение `NULL`
`property Modified: Boolean;`	Признак редактирования (`True`) текущей записи
`procedure Refresh;`	Обновление набора данных
`property State: TDataSetState;`	Информация о текущем состоянии набора

Состояния набора данных

В процессе своего функционирования (от открытия методом `Open` и до закрытия методом `Close`) набор данных может выполнять разнообразные операции: перемещение по записям, редактирование данных, вставку или удаление записей, поиск по различным параметрам и т. д.

В любой момент времени набор данных находится в некотором состоянии, т. е. подготовлен к выполнению действий строго определенного характера. Для каждой группы операций набор данных выполняет ряд подготовительных действий.

Все состояния набора данных делятся на две группы.

- состояния, в которые набор данных переходит автоматически, а также непродолжительные по времени состояния, сопровождающие функционирование полей набора данных (табл. 4.19);
- состояния, которыми можно управлять из приложения (табл. 4.20).

Таблица 4.19. **Автоматические состояния набора данных**

Константа состояния	Описание
`dsNewValue`	Включается при обращении к свойству `NewValue` поля набора данных
`dsOldValue`	Включается при обращении к свойству `OldValue` поля набора данных
`dsCurValue`	Включается при обращении к свойству `CurValue` поля набора данных
`dsInternalCalc`	Включается при расчете значений полей, для которых `FindKind = fkInternalCalc`
`dsCalcFields`	Включается при выполнении метода `OnCalcFields`
`dsBlockRead`	Включается механизм ускоренного перемещения по набору данных
`dsOpening`	Существует при открытии набора данных методом `Open` или свойством `Active`
`dsFilter`	Включается при выполнении метода `OnFilterRecord`

Текущее состояние набора данных передается в свойство `State`. Для управления состояниями набора данных используются методы `Open`, `Close`, `Edit`, `Insert`.

Рассмотрим, как изменяется состояние набора данных при выполнении основных операций:

- закрытый набор данных всегда имеет неактивное состояние `dsInactive`;

Таблица 4.20. **Управляемые состояния набора данных**

Константа состояния	Метод	Описание
dsInactive	Close	Набор данных закрыт
dsBrowse	Open	Данные доступны для просмотра, но недоступны для редактирования
dsEdit	Edit	Данные можно редактировать
dsInsert	Insert	К набору данных можно добавлять новые записи
dsSetKey	SetKey	Включается механизм поиска по ключу. Также могут использоваться диапазоны

- при открытии набор данных переходит в состояние просмотра данных dsBrowse. В этом состоянии можно перемещаться по записям набора данных и просматривать их содержимое, но редактировать данные нельзя;
- при необходимости редактирования данных набор должен быть переведен в состояние редактирования dsEdit (метод Edit). При перемещении на другую запись набор данных автоматически переходит в состояние просмотра;
- для того чтобы вставить в набор данных новую запись, необходимо использовать состояние вставки dsInsert (метод Insert). Новая пустая запись добавляется в набор данных по месту текущего курсора. При переходе на другую запись набор данных возвращается в состояние просмотра.

Поиск по произвольным полям

Для поиска по произвольной выборке полей можно использовать методы Locate и Lookup.

Метод Locate. Синтаксис вызова метода следующий:

```
function Locate (const KeyFields: string;
  const KeyValues: Variant; Options: TLocateOptions): Boolean;
```

Параметр KeyFields определяет список полей, по которым будет идти поиск (имена полей разделяются точкой с запятой).

Параметр KeyValues определяет список значений полей для поиска (значения разделяются запятой).

Параметр Options задает настройки поиска:

- опция loCaseinsensitive отключает проверку на регистр символов;

- опция loPartialKey включает поиск с минимальными отличиями.

В случае успеха поиска курсор набора данных устанавливается на найденной записи, а метод возвращает значение True.

Метод Lookup. Синтаксис вызова метода следующий:

```
function Lookup(const KeyFields: string; const KeyValues:
                    Variant;
const ResultFields: string): Variant;
```

Параметр KeyFields определяет список полей, по которым будет идти поиск (имена полей разделяются точкой с запятой).

Параметр KeyValues определяет список значений полей для поиска (значения разделяются запятой).

Параметр ResultFields содержит список имен полей, для которых функция (в случае успешного завершения) возвращает массив значений.

Фильтры. В набор данных встроен механизм фильтрации данных. Применение фильтра основано на следующих свойствах:

Filter — строка символов, содержащая текст фильтра;

Filtered — логическое значение, включающее и отключающее фильтр;

FilterOptions — параметры фильтра.

Фильтры можно разделить на статические и динамические.

Статические фильтры создаются во время разработки.

Динамические фильтры можно создавать и редактировать во время выполнения приложения, используя свойство Filter.

При создании текста фильтра для свойства Filter используются имена полей соответствующей таблицы, а для задания отношений применяются операторы сравнения (>, >=, <, <=, =, <>), связывающие операнд — имя поля с операндом-значением, и логические операторы (and, or, not), объединяющие условия отбора для нескольких полей, например:

```
Field1 > 100 AND Field2 = 'лекция'
```

Фильтр начинает работать только после того, как свойству Filtered присваивается истинное значение. Перед изменением текста динамического фильтра или для отключения фильтра свойству Filtered присваивается значение False.

Параметры фильтра могут принимать следующие значения:

`foCaseInsensitive` — отключает сравнение строковых значений с учетом регистра символов;

`foNoPartialCompare` — отключает отбор строковых значений по части строки.

4.4.2. Разработка приложений доступа к внешним источникам данных

В основе процесса разработки лежит триада компонентов:

- невизуальные компоненты набора данных;
- невизуальные компоненты `TDataSource`;
- визуальные компоненты отображения данных.

Модуль данных

Для размещения компонентов доступа к данным в приложении баз данных рекомендуется использовать специальную форму — модуль данных (класс `TDataModule`). Для создания модуля данных можно воспользоваться Репозиторием объектов (пиктограмма модуля данных **Data Module** расположена на странице **New**) или функцией главного меню Delphi **File→New→Data Module**.

В модуле данных можно размещать только невизуальные компоненты. Так как класс `TDataModule` происходит непосредственно от класса `TComponent`, у него почти полностью отсутствуют свойства и методы-обработчики событий.

Модуль данных доступен разработчику, как и любой другой модуль проекта, на этапе разработки. Пользователь приложения не может увидеть модуль данных во время выполнения.

Модули проекта, обращающиеся к компонентам доступа к данным, расположенным в модуле данных, должны содержать имя модуля в секции `uses`:

```
unit InterfaceModule;
...
implementation
uses DataModule1;
```

Изменение значения любого свойства компонентов доступа к данным, размещенных в модуле данных, проявится сразу же во всех модулях, к которым подключен модуль данных.

Подключение набора данных

Компонент доступа к данным является основой организации обмена данными с источником данных. В процессе работы приложения такой компонент взаимодействует с функциями соответствующей технологии доступа к данным. Все компоненты доступа к данным являются невизуальными.

Для обеспечения доступа к таблице источника данных необходимо (после размещения компонента, инкапсулирующего набор данных, на форме модуля данных приложения) выполнить следующие действия:

1. *Подключить компонент к источнику данных.* Для этого в зависимости от конкретной технологии используется специальный компонент, устанавливающий соединение.

2. *Подключить к компоненту таблицу источника данных.* После выполнения действий первого этапа в списке свойства `TableName`, доступного в Инспекторе объектов, должны появиться имена всех доступных в подключенном источнике данных таблиц. После выбора имени таблицы в свойстве `TableName` компонент связывается с ней.

3. *Активизировать связь между компонентом и таблицей источника данных.* Связь активизируется, если свойству `Active` присвоить значение `True`. Значение свойства может быть установлено прямым присваиванием при проектировании (в Инспекторе объектов) или в исходном коде приложения, а также с помощью метода `Open` или `Close`, который, соответственно, открывает набор данных или закрывает его.

Настройка компонента TDataSource

Следующий этап обеспечения работы с источником данных — размещение на форме и настройка компонента `TDataSource`, который должен обеспечить взаимодействие набора данных с компонентами отображения данных. Каждому набору данных соответствует свой (один или несколько) компонент `TDataSource`.

Чтобы связать набор данных и компонент `TDataSource`, необходимо заполнить (например, в Инспекторе объектов) свойство `DataSet` компонента `TDataSource`, служащее указателем на экземпляр компонента доступа к данным. В Инспекторе объектов в списке этого свойства перечислены все доступные компоненты наборов данных. В табл. 4.21 приведены некоторые свойства и методы компонента.

Таблица 4.21. **Свойства и методы компонента TDataSource**

Свойство/метод	Описание
`property DataSet: TDataSet;`	Имя связанного компонента набора данных
`property State: TDataSetState;`	Текущее состояние набора данных (активен, неактивен, находится в режиме просмотра, редактирования, вставки и т. п.)
`property Enabled: Boolean;`	Подключение или отключение всех связанных визуальных компонентов. При значении `False` ни один связанный компонент отображения данных не будет работать
`property AutoEdit: Boolean;`	Автоматический перевод набора данных в режим редактирования при получении фокуса одним из связанных визуальных компонентов (значение `True`)
`procedure Edit;`	Переводит связанный набор данных в режим редактирования
`function IsLinkedTo(DataSet: TDataSet): Boolean;`	Возвращает значение `True`, если компонент, указанный в параметре `DataSet`, действительно связан с данным компонентом `TDataSource`

Отображение данных

Интерфейс, обеспечивающий работу пользователя с источником данных, строится на основе компонентов отображения данных, специально предназначенных для решения задач просмотра и редактирования данных.

Компоненты отображения данных должны быть связаны с компонентом `TDataSource` и через него — с компонентом набора данных. Имя компонента `TDataSource` указывается в свойстве `DataSource`, которое присутствует во всех компонентах отображения данных. Один компонент отображения данных можно связать только с одним компонентом `TDataSource`.

Компоненты отображения данных можно разделить на следующие группы:

- компоненты для отображения данных одного поля (в таких компонентах имеется свойство `DataFieid`, которое определяет отображаемое поле связанного набора данных);
- компоненты для отображения набора данных (например, компонент `TDBGrid` представляет данные в виде таблицы,

в столбцах которой размещаются поля набора данных, а в строках — записи);

- компоненты для обеспечения навигации по данным (`TDBNavigator` — предназначен для перемещения по записям набора данных).

4.4.3. Пример работы с Рабочей книгой Excel

Рассмотрим пример работы с Рабочей книгой процессора электронных таблиц Microsoft Excel из приложения Delphi.

Пусть необходимо обеспечить доступ к таблице расписания занятий, подготовленной в среде Excel (рис. 4.32)[1]. Продемонст-

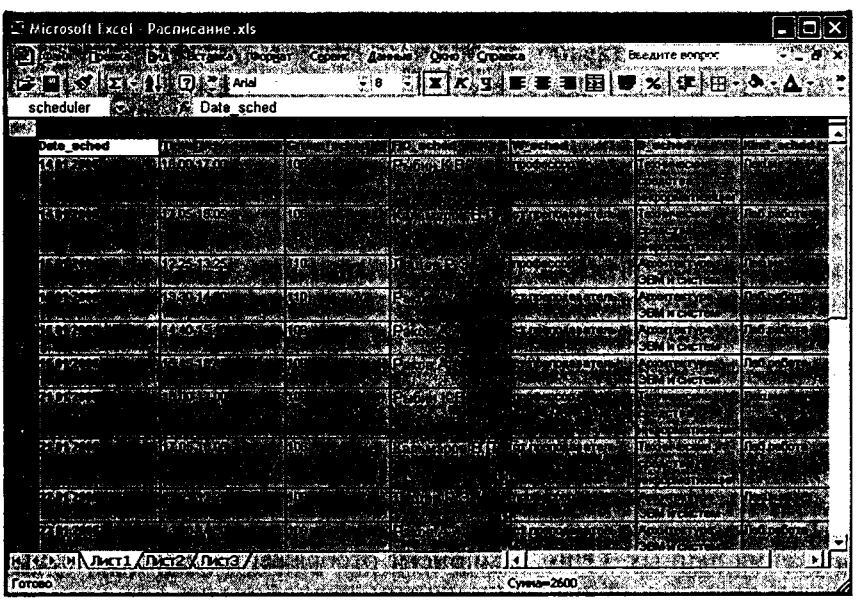


Рис. 4.32. Расписание занятий в рабочей книге Excel

[1] Для представления множества ячеек листа рабочей книги Excel как таблицы внешнего источника данных необходимо выполнить функцию присвоения имени выделенному блоку ячеек (команда главного меню **Вставка → Имя**). В рассматриваемом примере блоку ячеек расписания присвоено имя scheduler.

рируем основные этапы разработки приложения и реализацию простейших функций отбора записей по заданным критериям.

Выбор технологии доступа и соединение с источником данных

Для обеспечения доступа к источнику данных воспользуемся возможностями технологии ADO и разместим на форме модуля данных компонент `TADOConnection` (страница **ADO** палитры компонентов).

Настройка соединения состоит в заполнении свойства `ConnectionString` компонента в Инспекторе объектов. На рис. 4.33 последовательно представлены этапы формирования строки соединения в редакторе **Data Link Properties**.

На первой странице редактора соединения необходимо указать механизм соединения — провайдера данных. Для Excel в технологии ADO это **Microsoft Jet 4.0 OLE DB Provider** (рис. 4.33, *а*).

На второй странице указывается имя файла электронной таблицы (рис. 4.33, *б*). Для обеспечения работы с файлами форматов последних версий продукта необходимо на последней странице изменить значение свойства **Extended Properties** на **Excel 8.0** (рис. 4.33, *в*).

Связь источника данных с приложением обеспечена, если проверка подключения выполнена успешно.

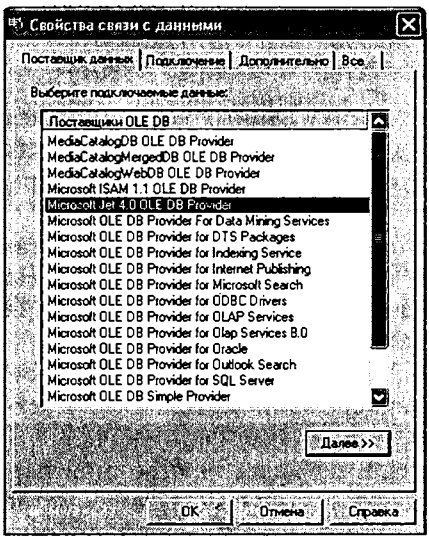
Подключение набора данных

Для соединения с таблицей источника данных разместим на форме модуля данных компонент `TADOTable` (страница **ADO** палитры компонентов) и заполним свойства `Connection` и `TableName`.

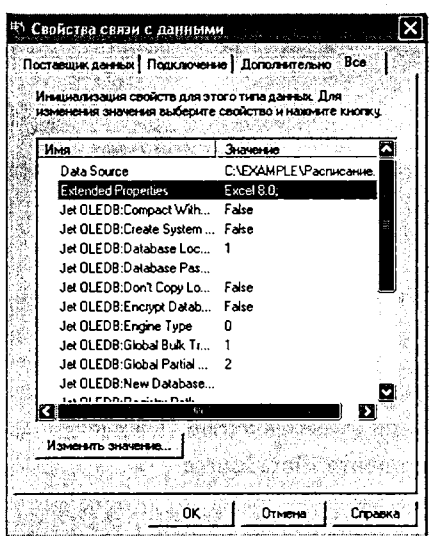
Раскрывающийся список допустимых значений свойства `Connection` содержит только одно возможное соединение — `ADOConnection1`, а в раскрывающемся списке значений свойства `TableName` необходимо выбрать таблицу `scheduler` (рис. 4.34).

Настройка компонента TDataSource

Компонент `TDataSource` расположен на странице **Data Access** палитры компонентов. Раскрывающийся список возможных значений свойства `DataSet` содержит только имя компонента набора данных `ADOTable1` (рис. 4.35).



а *б*



в

Рис. 4.33. Настройка соединения с источником данных

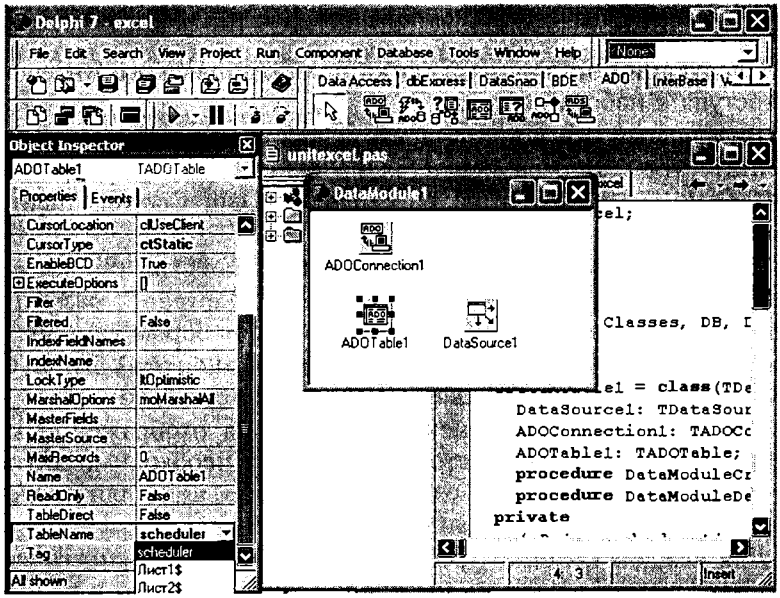


Рис. 4.34. Связь источника данных с набором данных

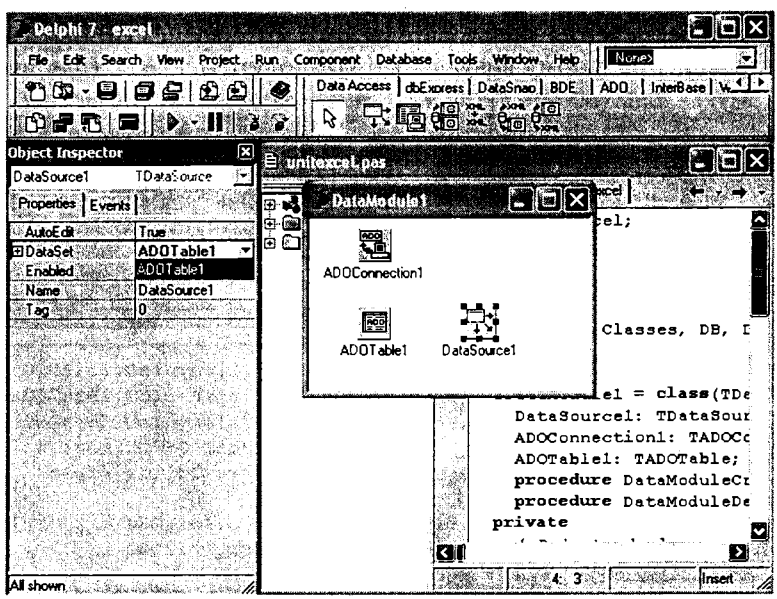


Рис. 4.35. Настройка компонента TDataSource

Размещение визуальных компонентов

Следующий этап — разработка пользовательского интерфейса доступа к набору данных.

Визуальный компонент, представляющий набор данных как таблицу (`TDBGrid`), размещается на основной форме приложения. Компонент находится на странице **Data Controls** палитры компонентов.

В раскрывающемся списке значений свойства `DataSource` необходимо выбрать имя компонента `DataSource1`, размещенного на форме модуля данных, для связи с набором данных. Таким образом, посредством компонента `DataSource1` будет осуществляться отображение данных таблицы `scheduler`.

Для настройки компонента `TDBGrid` на отображение конкретных полей таблицы источника данных необходимо активизировать в Инспекторе объектов свойство `Columns` для последовательного определения колонок.

Для каждой колонки требуется обязательно заполнить свойство `FieldName` — указать имя поля, отображаемого в колонке. Для создания нового интерфейсного имени поля — заголовка колонки — необходимо заполнить свойство `Caption` (рис. 4.36).

Для обеспечения пользовательского интерфейса отбора записей добавим на форму визуальные компоненты `TEdit` (для ввода

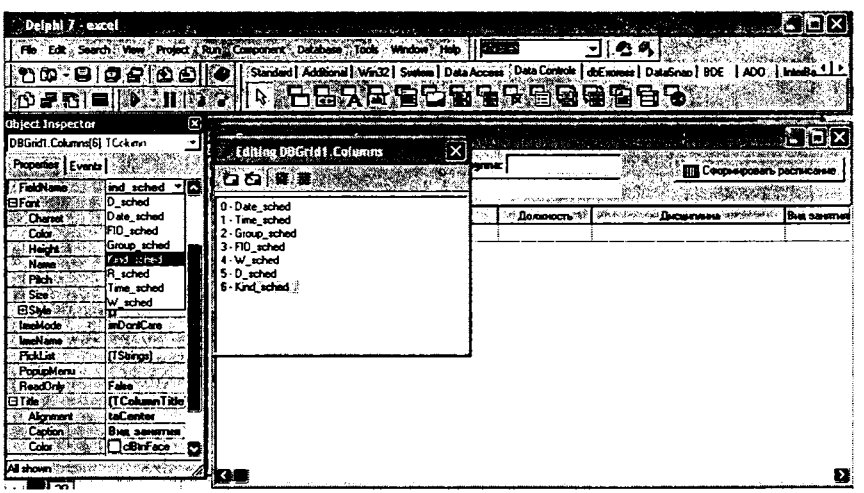


Рис. 4.36. Настройка полей таблицы

поисковых значений — фамилии преподавателя, группы и дисциплины) и кнопку, управляющую запуском поисковых операций.

Программирование обработчиков событий

В завершение необходимо определить момент соединения с источником данных и разработать процедуру формирования и запуска фильтров.

Для непосредственной связи с источником данных и отсоединения от него используем события формы модуля данных `OnCreate` и `OnDestroy`:

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);
begin
try
ADOConnection1.Connected := true;
ADOTable1.Open;
except
ShowMessage('Ошибка соединения!')
end;
end;

procedure TDataModule1.DataModuleDestroy(Sender: TObject);
begin
ADOConnection1.Connected := false;
end;
```

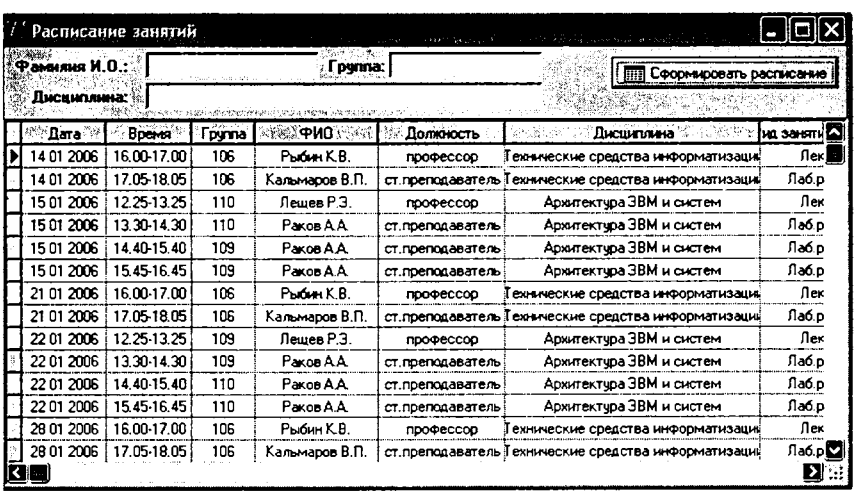
Создание и запуск фильтров должно осуществляться по нажатию на кнопку **Сформировать расписание** (компонент `BitBtn1`) и может быть запрограммировано в обработчике события `OnClick`:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var sfilter: string;
begin
DBGrid1.Columns[2].Visible := true;
          // Колонки 2–5 становятся видимыми
DBGrid1.Columns[3].Visible := true;
DBGrid1.Columns[4].Visible := true;
DBGrid1.Columns[5].Visible := true;
if trim(EdFIO.Text)<> '' then
  begin sfilter := sfilter + 'FIO_sched = ''' +
                                  EdFIO.Text + '''';
        DBGrid1.Columns[3].Visible := false;
        DBGrid1.Columns[4].Visible := false;
          // Если есть значение для поиска –
          // колонки становятся невидимыми
        end;
```

```
if trim(EdD.Text)<> '' then
  begin if sfilter <> '' then sfilter := sfilter + ' and ';
        sfilter := sfilter + 'D_sched = ''' + EdD.Text + '''';
        DBGrid1.Columns[5].Visible := false;
        end;
if trim(EdGroup.Text)<> '' then
  begin if sfilter <> '' then sfilter := sfilter + ' and ';
        sfilter := sfilter + 'Group_sched = ''' +
         EdGroup.Text + '''';
        DBGrid1.Columns[2].Visible := false;
        end;
datamodule1.ADOTable1.Filter := sfilter;
datamodule1.ADOTable1.Filtered := true;
end;
```

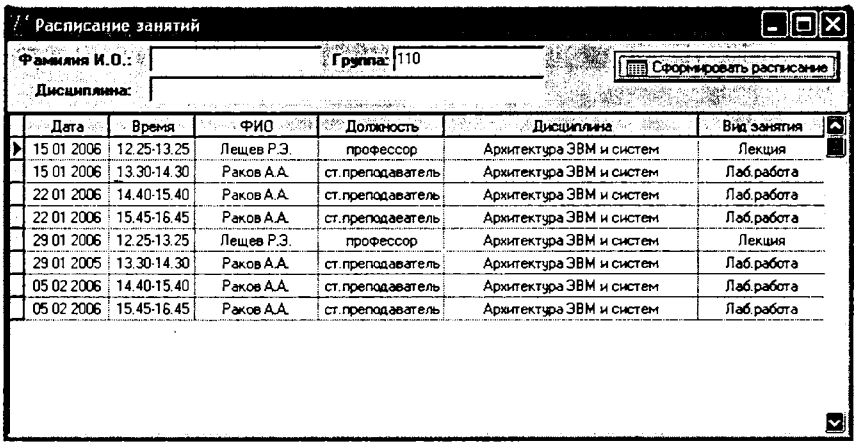
При разработке процедуры был использован принцип, согласно которому столбцы, содержащие поисковые значения, не включаются в результат отбора.

Итак, построено приложение, обеспечивающее отображение и простейшую обработку данных внешнего источника данных. При всей своей простоте приведенный пример тем не менее иллюстрирует технологию связи приложения с источником данных и порядок размещения и привязки компонентов графического интерфейса при решении подобных задач. На рис. 4.37 и 4.38 представлены примеры работающего приложения.



Дата	Время	Группа	ФИО	Должность	Дисциплина	Вид занятий
14.01.2006	16.00-17.00	106	Рыбин К.В.	профессор	Технические средства информатизации	Лек
14.01.2006	17.05-18.05	106	Кальмаров В.П.	ст.преподаватель	Технические средства информатизации	Лаб.р
15.01.2006	12.25-13.25	110	Лещев Р.Э.	профессор	Архитектура ЭВМ и систем	Лек
15.01.2006	13.30-14.30	110	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.р
15.01.2006	14.40-15.40	109	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.р
15.01.2006	15.45-16.45	109	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.р
21.01.2006	16.00-17.00	106	Рыбин К.В.	профессор	Технические средства информатизации	Лек
21.01.2006	17.05-18.05	106	Кальмаров В.П.	ст.преподаватель	Технические средства информатизации	Лаб.р
22.01.2006	12.25-13.25	109	Лещев Р.Э.	профессор	Архитектура ЭВМ и систем	Лек
22.01.2006	13.30-14.30	109	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.р
22.01.2006	14.40-15.40	110	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.р
22.01.2006	15.45-16.45	110	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.р
28.01.2006	16.00-17.00	106	Рыбин К.В.	профессор	Технические средства информатизации	Лек
28.01.2006	17.05-18.05	106	Кальмаров В.П.	ст.преподаватель	Технические средства информатизации	Лаб.р

Рис. 4.37. Окно программы при старте



Дата	Время	ФИО	Должность	Дисциплина	Вид занятия
15 01 2006	12.25-13.25	Лещев Р.Э.	профессор	Архитектура ЭВМ и систем	Лекция
15 01 2006	13.30-14.30	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.работа
22 01 2006	14.40-15.40	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.работа
22 01 2006	15.45-16.45	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.работа
29 01 2006	12.25-13.25	Лещев Р.Э.	профессор	Архитектура ЭВМ и систем	Лекция
29 01 2005	13.30-14.30	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.работа
05 02 2006	14.40-15.40	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.работа
05 02 2006	15.45-16.45	Раков А.А.	ст.преподаватель	Архитектура ЭВМ и систем	Лаб.работа

Рис. 4.38. Результаты поиска

Упражнения

1. Модифицируйте приложение «Редактор текста», добавив возможность изменения через Главное меню шрифта выделенного в окне редактирования текста.

2. Модифицируйте приложение «Редактор текста», добавив в меню сервисные функции:
   - ликвидация лишних пробелов между словами;
   - ликвидация повторяющихся знаков препинания (две точки подряд, две запятые подряд и т. п.);
   - проверка правильности написания первого слова в предложении (обязательно с прописной буквы);
   - преобразование всех букв выделенного текста в прописные (строчные).

3. Дополните меню приложения «Редактор текста» функциями сбора статистики по отображенному в окне редактирования тексту:
   - подсчет количества символов;
   - подсчет количества слов;
   - подсчет количества предложений.

4. Модифицируйте приложение — пример работы с Рабочей книгой Excel, дополнив его функцией подсчета нагрузки (количества часов занятий) для заданного преподавателя (всего и отдельно по видам занятий).

5. Модифицируйте приложение — пример работы с Рабочей книгой Excel, дополнив его функцией вывода всех занятий в текущем месяце.

Контрольные вопросы

1. Постройте сравнительную таблицу операторных возможностей для языков Java, Object Pascal.
2. Перечислите и охарактеризуйте базовые типы данных языка Object Pascal.
3. Перечислите и охарактеризуйте структурированные типы данных языка Object Pascal.
4. Охарактеризуйте особенности типа данных Variant.
5. Сформулируйте различия между короткими и длинными строками в Object Pascal.
6. Охарактеризуйте реализацию принципов ООП в Object Pascal.
7. Перечислите и охарактеризуйте разновидности условных операторов языка Object Pascal.
8. Перечислите и охарактеризуйте разновидности операторов цикла языка Object Pascal.
9. Охарактеризуйте действие стандартных процедур Break и Continue в операторах цикла.
10. Охарактеризуйте возможность использования в качестве параметров процедур и функций массивов открытого типа.
11. Охарактеризуйте механизм обработки исключительных ситуаций в Object Pascal.
12. Перечислите и охарактеризуйте основные интерфейсные окна среды Delphi.
13. Перечислите и охарактеризуйте файлы, входящие в состав проекта Delphi.
14. Поясните, как выполняются компиляция, запуск и отладка приложения в среде Delphi.
15. Назовите события, общие для всех визуальных компонентов.
16. Определите понятие «Источник данных».
17. Перечислите и охарактеризуйте механизмы, составляющие общую архитектуру приложения, работающего с источниками данных.
18. Дайте определение набора данных.
19. Охарактеризуйте последовательность действий по подключению набора данных к источнику данных.
20. Охарактеризуйте методы `Locate` и `Lookup`.

Список литературы

1. *Агафонов В. Н.* Спецификация программ: понятийные средства и их организация. Новосибирск: Наука, 1987.

2. *Архангельский А. Я.* Object Pascal в Delphi. М.: Бином, 2002.

3. *Баженова И. Ю.* Язык программирования Java. М.: Диалог-МИФИ, 1997.

4. Базисный Рефал и его реализации на вычислительных машинах: методические рекомендации. М.: ЦНИПИАСС, 1977.

5. *Братко И.* Программирование на языке Пролог для искусственного интеллекта: пер. с англ. М.: мир, 1990.

6. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином; СПб.: Невский диалект, 1998.

7. *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя. М.: ДМК, 1998.

8. *Ван Тассел Д.* Стиль, разработка, эффективность, отладка и испытание программ. М.: Мир, 1981.

9. *Вендров А. М.* Проектирование программного обеспечения экономических информационных систем. М.: Финансы и статистика, 2000.

10. *Вирт Н.* Алгоритмы и структуры данных: пер. с англ. М.: Мир, 1989.

11. *Голицына О. Л., Попов И. И.* Основы алгоритмизации и программирования: учеб. пособие. М.: ФОРУМ: ИНФРА-М, 2006.

12. *Григас Г.* Начала программирования. М. Просвещение, 1987.

13. *Гультяев А. К., Машин В. А.* Проектирование и дизайн пользовательского интерфейса. СПб.: КОРОНА принт, 2000.

14. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование: пер. с англ. М.: Мир, 1975.

15. *Дарахвелидзе П. Г., Марков Е. П.* Программирование в Delphi 7. СПб.: БХВ-Петербург, 2003.

16. Информатика: учеб. пособие для пед. спец. высших учеб. завед. / А. Р. Есаян, В. И. Ефимов, Л. П. Лапицкая и др. М.: Просвещение, 1991.

17. *Каймин В. А.* Информатика: учебник. М.: ИНФРА-М, 2000.

18. *Кауфман В. Ш.* Языки программирования. Концепции и принципы. М.: Радио и связь, 1993.

19. *Костогрызов А. И., Липаев В. В.* Сертификация качества функционирования автоматизированных информационных систем. М.: «Вооружение. Политика. Конверсия», 1996.

20. *Кострикин А. И.* Введение в алгебру. М.: Наука, 1977.

21. *Липаев В. В.* Надежность программных средств. Сер. Информатизация России на пороге XXI века. М.: СИНТЕГ, 1998.

22. *Мендельсон Э.* Введение в математическую логику. М.: Наука, 1976.

23. *Михайлов В. Ю., Степанников В. М.* Современный Бейсик для IBM PC. Среда, язык, программирование. М.: Изд-во МАИ, 1993.

24. *Морозов В. П., Шураков В. В.* Основы алгоритмизации, алгоритмические языки и системы программирования: Задачник. М.: Финансы и статистика, 1994.

25. *Нагао М., Катаяма Т., Уэмура С.* Структуры и базы данных: пер. с япон. М.: Мир, 1986.

26. *Одинцов И. О.* Профессиональное программирование. Системный подход. СПб.: БХВ-Петербург, 2002.

27. *Орлов С. А.* Технологии разработки программного обеспечения: учебник. СПб.: Питер, 2002.

28. *Симкин С., Бартлетт Н., Лесли А.* Программирование на Java. Путеводитель: пер. с англ. К.: НИПФ «ДиаСофт Лтд.», 1996.

29. *Турчин В. Ф.* Метаязык для формального описания алгоритмических языков // Цифровая вычислительная техника и программирование. М.: Советское радио, 1966. С.116—119.

30. *Фридман А. Л.* Основы объектно-ориентированной разработки программных систем. М.: Финансы и статистика, 2000.

31. *Хомоненко А. Д.* и др. Delphi 7 / под общ. ред. А. Д. Хомоненко. СПб.: БХВ-Петербург, 2003.

32. *Хьюз Дж., Мичтом Дж.* Структурный подход к программированию: пер. с англ. М.: Мир, 1980.

33. *Черемных С. В., Семенов И. О., Ручкин В. С.* Структурный анализ систем: IDEF-технологии. М.: Финансы и статистика, 2001.

34. Dictionary of Computing (Data Communication, Hardware and Software Basics, Digital Electronic) Ed. by Frank J. Galland, John Wiley & Sons, Datology Press Ltd, Windsor, England, 1983.

35. IEEE Std 1209—1992. IEEE Recommended Practice for the Evaluation and Selection of CASE Tools.

Глоссарий терминов и сокращений

Ada — язык высокого уровня для программирования сложных комплексов программ. Обеспечивает возможности строгого определения типов данных. Ориентирован на создание встроенных систем реального времени, распределенных систем, высоконадежных комплексов программ и повторно используемых компонентов.

ALGOL — алгоритмический язык высокого уровня, разработанный для решения научных и инженерных вычислительных задач, в котором впервые (1960 г.) были определены и стандартизованы основные понятия и объекты языков программирования (ALGOrthmic Language).

ASCII (American Standard Code for Information Interchange) — американский стандартный код для обмена информацией: соглашение для представления символьной информации; код для представления английской текстовой информации, используемый с отдельными модификациями в большинстве вычислительных систем.

Basic — «многоцелевой язык символических команд для начинающих». Самый простой язык высокого уровня для освоения принципов алгоритмизации и программирования. Для многих мини- и микро-ЭВМ предназначался в качестве единственного языка высокого уровня.

BPwin — инструмент для моделирования бизнес-процессов. Позволяет оптимизировать деятельность организации: спроектировать организационную структуру, снизить издержки, исключить ненужные операции и повысить эффективность. BPwin поддерживает сразу три нотации моделирования: IDEF0, IDEF3 и DFD.

C (Си) — универсальный язык программирования высокого уровня, предназначенный для создания ПС различных классов, включая операционные системы, инструментальные ПС, а также ПС деловых и научных применений.

CASE — программные средства, поддерживающие процессы создания и сопровождения ИС, включая анализ и формулировку требований, проектирование прикладного ПО (приложений) и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

COBOL — алгоритмический язык высокого уровня, ориентированный на обработку данных при решении экономических и управленческих задач (COmmon Buisiness Oriened Language).

COM (Component Object Model) — составляющая программного обеспечения, поддерживающая OLE 2.

CP-437 — стандарт IBM для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для греческого алфавита.

CP-850 — стандарт IBM для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для восточно-европейских алфавитов.

CP-852 — стандарт IBM для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для греческого алфавита.

CP-862 — стандарт IBM для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для иврита.

CP-866 — стандарт IBM для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для русской кириллицы.

DFD (Data Flow Diagrams — диаграмма потоков данных) — средство структурного моделирования, представляет систему как набор функциональных процессов, связанных потоками данных. Цель такого представления — продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

ERwin — средство моделирования баз данных и хранилищ данных. Позволяет проектировать, документировать и сопровождать базы данных различных типов.

ERwin Examiner — инструмент для проверки структуры баз данных и создаваемых в ERwin моделей, позволяющий выявлять ошибки проектирования.

ERwin Modeling Suite — пакет, включающий CASE-средства ERwin, BPwin, ERwin Examiner и Paradigm Plus. Позволяет моделировать всевозможные виды деятельности, управлять изменениями, проектами и бизнес-процессами.

ESC-последовательность (Escape code) — коды разметки документов, аппаратурно интерпретируемые принтерами Epson (и др.), позволяющие выделять шрифты, размеры и стили печатаемых символов.

FORTRAN — первый из известных машинонезависимых языков программирования, ориентированный на научные и инженерные расчеты.

GUI (Graphical User Interface) — графический интерфейс пользователя.

IDEF (Integrated DEFinition) — семейство стандартов, определяющее взаимную совокупность методик и моделей концептуального проектирования. Разработано в США по программе Integrated Computer-Aided Manufacturing.

IDEF0 — метод моделирования, является частью семейства стандартов IDEF. Реализует методику функционального моделирования сложных систем. Рекомендуется для начальных стадий проектирования систем управления, производства, бизнеса и т. п., объединяющих людей, оборудование, программное обеспечение. Позволяет формулировать ответы на вопрос: «Что делает система?».

IDEF3 — метод моделирования, является частью семейства стандартов IDEF. Реализует поведенческое моделирование. Позволяет формулировать ответы на вопросы: «Как система выполняет функцию?». Предоставляет инструментарий для наглядного представления и моделирования сценариев автоматизируемых процессов.

Internet — всемирная компьютерная сеть; сеть сетей, объединяющая множество компьютерных сетей во всем мире и предоставляющая доступ к мировым информационным ресурсам.

ISO (International Organization for Standardization) — международная организация по стандартизации, МОС.

Java — объектно-ориентированный машинно-независимый язык программирования для Internet.

JavaScript — объектно-ориентированный язык программирования сценариев просмотра ресурсов WWW, является развитием HTML.

Latin-1 — международный стандарт (ISO-8859-1) для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для латиницы.

Latin-8 — международный стандарт (ISO-8859-8) для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для иврита.

Latin-C — международный стандарт (ISO-8859) для интерпретации 2-й половины (128—256) кода ASCII, таблица предназначена для кириллицы.

ModelMart — среда для работы группы разработчиков на ERwin и BPwin. Обеспечивает совместный доступ и редактирование моделей, является интегрирующим звеном для ERwin и BPwin.

Object Pascal — объектно-ориентированное расширение языка программирования Pascal. Базовый язык программирования в интегрированной среде разработки Delphi.

ODBC (Open Database Connectivity) — стратегический интерфейс Microsoft для вызова данных в гетерогенной среде реляционных и нереляционных систем управления базами данных. ODBC предназначен обеспечить универсальный набор команд интерфейса для доступа к данным.

OLE 2.0 (Object Linking and Embedding 2.0) — набор стандартных спецификаций и способов их реализации, находящийся в собственности и поддерживаемый Microsoft для составных документов.

Oracle Designer (входит в Oracle9i Developer Suite) — многофункциональное средство проектирования программных систем и баз данных.

Paradigm Plus 4 — CASE-средство для моделирования компонентов программного обеспечения и генерации объектного кода приложений на основе созданных моделей. Продукт можно использовать как при создании новых приложений, так и при изменении или объединении существующих.

Pascal — универсальный язык программирования высокого уровня, являющийся развитием языка ALGOL, ориентированный на широкий класс задач обработки данных и организации вычис-

лений (научные, инженерные, экономические, управленческие и пр.).

Prolog (ПРОЛОГ) — язык логического программирования, основанный на логике дизъюнктов Хорна. Будучи декларативным языком программирования, Prolog воспринимает в качестве программы некоторое описание задачи и производит поиск решения, пользуясь методом резолюций.

Rational Rose — средство моделирования объектно-ориентированных информационных систем, базирующееся на языке моделирования UML.

SQL — стандартизованный язык запросов к реляционной (табличной) базе данных, обеспечивающий реляционно полный набор операций на данными.

UML (Unified Modeling Language) — унифицированный язык моделирования, представляет собой язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы.

Агрегат данных — именованная совокупность элементов данных, представленных простой (векторной) или иерархической (группы или повторяющиеся группы) структурой. Примеры — массивы, записи, комплексные числа и пр.

Алгоритм — понятное и точное предписание (указание) исполнителю совершить определенную конечную последовательность действий, направленных на достижение указанной цели или решение поставленной задачи (приводящую от исходных данных к искомому результату).

Алгоритм Маркова (или ***нормальный алгоритм***) преобразует слова, заданные в некотором алфавите, на основе непосредственного доступа к различным частям слова. Представляет собой упорядоченный набор *формул подстановки* — пар слов, соединенных между собой стрелками двух видов: замена левой части формулы на правую и замена левой части формулы на правую и останов (принудительное завершение работы алгоритма).

Аппликативный (функциональный) язык программирования основан на функциональном подходе к программированию. Язык рассматривается с точки зрения нахождения функции, необходи-

мой для перевода памяти ПК из одного состояния в другое. Программа представляет собой набор функций, применяемых к начальным данным, позволяющий получить требуемый результат.

Арифметические операции — четыре действия арифметики (+, -, *, /) и операция получения остатка от деления (%) образуют группу арифметических операций. Их выполнение не имеет каких-либо особенностей, кроме как преобразование типов переменных при их несовпадении. Если в одном выражении встречаются переменные разных типов, как правило, осуществляется преобразование (приведение) типов.

Архитектура ЭВМ — общее описание структуры и функций ЭВМ на уровне, достаточном для понимания принципов работы и системы команд ЭВМ. Архитектура не включает в себя описание деталей технического и физического устройства компьютера. Основные компоненты архитектуры ЭВМ: процессор, внутренняя (основная) память, внешняя память, устройства ввода, устройства вывода.

Атрибут — поле данных, содержащее информацию об объекте.

Байт — машинное слово минимальной размерности, адресуемое в процессе обработки данных. Размерность байта — 8 бит — принята не только для представления данных в большинстве компьютеров, но и в качестве стандарта для хранения данных на внешних носителях, для передачи данных по каналам связи, для представления текстовой информации. Размерность всех форм представления данных устанавливается кратной байту. При этом машинное слово считается разбитым на байты, которые нумеруются, начиная с младших разрядов.

Библиотека (library) — набор функций, в том числе из стандартных библиотек, предопределенных переменных и констант, которые могут быть использованы в программе и хранятся в откомпилированном виде.

Блок — составной оператор, включающий описания переменных в начале. Это собственные переменные блока, действие которых не распространяется за его пределы, а время существования совпадает со временем его выполнения.

Валидация — автоматическая проверка распознанных данных на соответствие заданным правилам. Например, проверка на попадание численных данных в определенный интервал, проверка

совпадение сумм, указанных цифрами и прописью, проверка на соответствие формату или заданному значению.

Внешняя ссылка — обращение к переменной или вызов функции во внутреннем представлении модуля, которая определена в другом модуле и отсутствует в текущем.

Время (Time) — тип данных, предназначенный для отображения моментов событий, предполагает наличие встроенных или эмулируемых команд специальной арифметики.

Время выполнения (run time) — период, во время которого происходит выполнение программы.

Время компиляции (compiler time) — период, во время которого происходит компиляция программы. Во время компиляции обнаруживаются синтаксические ошибки.

Вызов функции (call interface) — выполнение ее тела с заданными значениями формальных параметров.

Выражение — множество взаимосвязанных операций над переменными и константами и скобок «()», в котором результат одной операции является операндом другой.

Генерация кода — преобразование элементарных действий, полученных в результате лексического, синтаксического и семантического анализа программы, в некоторое внутреннее представление. Это могут быть коды команд, адреса и содержимое памяти данных, либо текст программы на языке Ассемблера, либо стандартизованный промежуточный код (например, P-код). В процессе генерации кода производится и его оптимизация.

Главная (внутренняя, оперативная) память компьютера представляет собой упорядоченную последовательность байтов или машинных слов (ячеек памяти), проще говоря — массив. Номер байта или слова памяти, через который оно доступно как из команд компьютера, так и во всех других случаях, называется адресом. Если в команде непосредственно содержится адрес памяти, то такой доступ к этому слову памяти называется прямой адресацией.

Глобальные переменные. Вне тела функции можно определить глобальные переменные. Этими переменными могут пользоваться все функции, следующие по тексту. Глобальные переменные представляют собой общие данные программы.

Данные — информация, обработанная и представленная в формализованном виде для дальнейшей обработки.

Дата (Date) — тип данных, предназначенный для обработки и отображения дат событий и другой аналогичной информации, предполагает наличие встроенных или эмулируемых команд специальной арифметики.

Двойная точность (Double) — тип числовых данных (с фиксированной или плавающей точкой), размещенный в двух машинных словах, требует наличия операций специальной арифметики.

Двойное слово — машинное слово двойной длины (двойное слово) — используется для увеличения диапазона представления целых чисел. Двойные слова обрабатываются либо отдельными командами процессора, либо программно (эмуляция).

Дизъюнкт Хорна — дизъюнкт с не более чем одним неотрицательным членом, т. е. логическое предложение, имеющее следующий общий вид записи: $P_0 \vee \neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n$, $n >= 0$, где P_i — атомарная формула (предикат).

Директива условной компиляции позволяет в зависимости от задания тех или иных условий компилировать или исключать из программы на стадии компиляции отдельные фрагменты кода.

Дополнительный код — беззнаковая форма представления чисел со знаком. В двоичной системе счисления дополнение каждой цифры выглядит как инвертирование двоичного разряда, т. е. замена 0 на 1 и наоборот. Если же знак числа представляется старшим разрядом машинного слова, то получается простой способ представления отрицательного числа: взять абсолютное значение числа в двоичной системе; инвертировать все разряды, включая знаковый; добавить 1.

Заголовок функции описывает «интерфейс» функции. В нем имеется имя функции, по которому она известна далее в программе.

Запись логическая — идентифицируемая (именованная) совокупность элементов или агрегатов данных, воспринимаемая прикладной программой как единое целое при обмене информацией с внешней памятью. Запись — это упорядоченная в соответствии с характером взаимосвязей совокупность полей (элементов) данных, размещаемых в памяти в соответствии с их типом.

Запись физическая — совокупность данных, которая может быть считана или записана как единое целое одной командой ввода-вывода.

Идентификатор — имя переменной, состоит из больших и маленьких латинских букв, цифр и знака «_» (подчеркивание) и начинается с буквы.

Императивное программирование (***операционное, процедурное*** программирование) — исторически первая методология программирования. При императивном подходе память вычислительной машины условно делится на две части: в одной хранятся данные в виде переменных, в другой — команды, которые изменяют содержимое переменных. Императивное программирование основано на алгоритмическом мышлении — при построении программы необходимо ясно представлять, какие действия и в какой последовательности будут проводиться при ее выполнении.

Инициализация — установка начальных значений во время трансляции.

Инкапсуляция (ООП) — свойство объектно-ориентированного программирования, состоящее в сокрытии информации об объекте и комбинировании в объекте данных и функций.

Интегрированная среда разработки приложений (Integrated Development Environment — IDE) помимо всех функций системы программирования содержит инструменты для упрощения конструирования графического интерфейса пользователя и большой спектр сервисов, включающих управление версиями проектов, просмотра и управления информацией, библиотеки классов (для поддержки объектно-ориентированной разработки), мастера создания шаблонов приложений, репозитории проекта и т. п.

Интерпретатор — разновидность транслятора, осуществляет покомандную расшифровку программы и выполнение инструкций входного языка (в среде конкретной системы программирования).

Интерфейс (Interface) — совокупность технических и программных средств визуализации информации и ввода данных, обеспечивающая интерактивное взаимодействие пользователя с системой.

Ключевая директива включает или отключает определенные директивой возможности компилятора.

Код ASCII (ASCII code — American Standart Code for Information Interchange) — 7- или 8-битовый код обмена данными; другие обозначения — IA-5, ANSI X.34, ISO-7 (код ISO-7 отличается

10-ю кодовыми комбинациями, зарезервированными для национальных применений).

Кодовая таблица (Code Page) — таблица, устанавливающая стандартизованное соответствие графических символов и бинарных кодов, определяемое применением (алфавит, программы, устройства ЭВМ).

Компилятор — разновидность транслятора, осуществляет подготовку результирующего (исполнительного) модуля, который может выполняться на ЭВМ практически независимо от среды.

Компоновщик, редактор связей, линкер (link, Linkage editor) осуществляет процесс сборки программы из объектных модулей, при котором производится объединение модулей в исполняемую программу и связывание вызовов внешних функций, расположенных в различных объектных модулях и библиотеках, и их внутреннего представления (кодов).

Корректность программы — характеристика программного средства, которая определена только в области изменения исходных данных, заданных требованиями спецификации, и не зависит от динамики функционирования программы в реальном времени. Степень некорректности программ тем самым определяется вероятностью попадания реальных исходных данных в область, которая задана требованиями спецификации и технического задания, но не была проверена при тестировании и испытаниях.

Косвенная адресация — случай, когда машинное слово содержит адрес другого машинного слова. Тогда доступ к данным во втором машинном слове через первое называется косвенной адресацией. Команды косвенной адресации имеются в любом компьютере и являются основой любого регулярного процесса обработки данных. Действительно, содержимое первого машинного слова можно формировать программно, работая с различными (например, последовательными расположенными) словами памяти.

Лексика языка программирования — правила «правописания слов» программы, таких как идентификаторы, константы, служебные слова, комментарии. Лексический анализ разбивает текст программы на указанные элементы. Особенность любой лексики — ее элементы представляют собой регулярные линейные последовательности символов..

Логические операции — операции И (&, AND), ИЛИ (|, OR) и НЕ (!, NOT), которые объединяют по правилам логики несколько условий в одно условное выражение. Благодаря тому, что любая логическая операция может быть представлена с помощью трех основных логических операций, набора элементов И, ИЛИ и НЕ в принципе достаточно для построения любого устройства процессора компьютера, а также для описания любых алгоритмов.

Логическое данное (Boolean) — тип данных, предназначенный для составления логических выражений и управления вычислительным процессом; типу соответствуют определенные операции и функции (логические).

Логическое программирование (ЛП) — методология программирования, основанная на формальной логике. Программа задается как набор логических утверждений. Путем применения операций унификации (сопоставления) и редукции (преобразования, упрощения) система находит решение задачи. Искомые величины задаются в виде переменных в логических отношениях и запросах. В ЛП, аналогично функциональному программированию, переменная есть символическое обозначение некой сущности. Значение переменной не может изменяться: оно либо найдено, либо нет.

Локальные переменные — переменные, которые «известны» только данной функции (действительны в данном блоке) и являются ее собственностью. Создаются в памяти при входе в тело функции и уничтожаются при выходе. Локальные переменные объявляются для функции, которой необходимо иметь собственные данные.

Массив — упорядоченная последовательность переменных одного и того же типа, имеющая общее имя. Номер элемента в последовательности называется *индексом*. Количество элементов в массиве не может быть изменено в процессе выполнения программы. Элементы массива размещаются в памяти последовательно и нумеруются от 1 до n, где n — их количество в массиве.

Машина Тьюринга — абстрактное вычислительное устройство, предложенное Аланом Тьюрингом для формального определения алгоритма.

Машинное слово — упорядоченное множество двоичных разрядов, используемое для хранения команд программы и обрабатывае-

мых данных. Каждый разряд, называемый битом — это двоичное число, принимающее значения только 0 или 1. Разряды в слове обычно нумеруются справа налево начиная с 0. Количество разрядов в слове называется размерностью машинного слова или его разрядностью.

Модель данных — базовый инструментарий, обеспечивающий на формальном абстрактном уровне конкретные способы представления объектов и связей.

Надежная программа — это программа, которая должна обеспечивать достаточно низкую вероятность отказа в процессе функционирования в реальном масштабе времени.

Наследование (ООП) — свойство объектно-ориентированного программирования, когда новый (производный) класс может быть определен на основе уже имеющегося (базового). При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки.

Объектно-ориентированное программирование (ООП) — методология программирования, использующая объектную декомпозицию, основанную на выделении объектов и связей между ними. В отличие от процедурного подхода к программированию, когда описание алгоритма решения некоторой задачи представляет собой последовательность действий, объектно-ориентированный подход предлагает описывать программные системы в виде взаимодействия объектов и, прежде всего, создать некоторый инструментарий, присущий решаемой задаче, а затем уже программировать в терминах этой задачи.

Объектный модуль — файл данных, содержащий оттранслированные во внутреннее представление собственные функции и переменные, а также информацию о внешних ссылках и точках входа модуля в символьном виде.

Операнд — переменная, константа, выражение, участвующие в операции. Унарная операция — операция с одним операндом. Бинарная — операция с двумя операндами.

Оператор — синтаксическая единица программы, которая отражает логику ее работы (последовательная, ветвящаяся, повторяющаяся). Для операторов характерен принцип вложенности: составными частями оператора могут быть любые другие операторы и сам он в свою очередь, может входить составной частью в оператор более высокого уровня.

Оператор break — производит выход из внутреннего цикла, то есть переходит к первому оператору, следующему за текущим оператором цикла. Заметим, что «покинуть»одновременно несколько вложенных друг в друга циклов с помощью break не удается.

Оператор continue выполняет переход из тела цикла к его повторяющейся части, т. е. досрочно завершает текущий шаг и переходит к следующему.

Оператор return формирует значение переменной-результата как значение выражения, стоящего за этим ключевым словом. Кроме того, он досрочно прекращает выполнение тела функции и возвращает программу в ту точку, где произошел вызов функции.

Оператор цикла обеспечивает повторяющееся выполнение следующего за ним оператора (или блока) — тела цикла. Шаг цикла (итерация цикла) — конкретный факт выполнения тела цикла.

Операции инкремента ++ и декремента -- (ЯП Си, Java) увеличивают или уменьшают значение единственного операнда до или после использования его значения в выражении.

Операции сравнения («=» — равно, «!=», «<>» — не равно, а также «>», «<», «>=», «<=») дают в качестве результата значения «истина» или «ложь». Выражения с такими значениями называются условными, поскольку обозначают выполнение или, наоборот, невыполнение некоторого условия в программе. Они используются в условном операторе (`if`) и в операторах цикла (`do-while`, `for`).

Определение переменной «создает» переменную, выделяя под нее память, задавая имя, тип и, возможно, начальное значение.

Переменная — именованная область памяти программы, в которой размещены данные с определенной формой представления (типом). Для того чтобы воспользоваться переменной, необходимо произвести некоторые предварительные действия — выпол-

нить определение переменной. Только при наличии такого определения транслятор будет знать ее имя, тип данных и, возможно, начальные значения.

Персональный компьютер (ПК, Personal Computer) — малогабаритная ЭВМ, обычно реализованная на микропроцессорах, использующая однопользовательскую операционную систему.

Подпрограмма (процедура, функция) — средство, позволяющее многократно использовать в разных местах основной программы один раз описанный фрагмент алгоритма.

Поле (Field) — однородный элемент данных определенного типа, описывающий аспект объекта или соответствующий фрагменту документа.

Полиморфизм (ООП) — свойство объектно-ориентированного программирования, основанное на присваивании действию одного имени, которое затем разделяется вверх и вниз по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, подходящим именно ему. Практический смысл полиморфизма заключается в том, что программист может сделать регулярным процесс обработки несовместимых объектов различных типов при наличии у них такого полиморфного метода.

Препроцессинг — предварительная фаза трансляции на уровне преобразования исходного текста программы с использованием директив препроцессора.

Приведение типов — преобразование типов операндов в операциях. В выражениях в качестве операндов могут присутствовать переменные и константы разных типов. Результат каждой операции также имеет свой определенный тип, который зависит от типов операндов. Если в бинарных операциях типы данных обоих операндов совпадают, то результат будет иметь тот же самый тип. Если нет, то транслятор должен включить в код программы неявные операции, которые преобразуют тип операндов, т. е. выполнить приведение типов. Преобразование типов может включать в себя следующие действия: увеличение или уменьшение разрядности машинного слова, т. е. «усечение» или «растягивание» целой переменной; преобразование целой переменной в переменную с плавающей точкой и наоборот; преобразование знаковой формы представления целого в беззнаковую и наоборот.

Прикладная программа (Application) — программное средство, предназначенное для решения определенного класса задач и поддерживающее некоторый класс технологий. Прикладная программа, подготовленная на языке программирования, обычно называется исходной программой (source program). Двоичная версия программы, выходящая из компилятора и размещаемая в оперативной памяти для последующего выполнения, называется исполнительным модулем (object module) в машинном представлении. Она представляет собой последовательность машинных инструкций (команд), которые конкретно указывают компьютеру, что надо делать.

Присваивание — операция, которая значение выражения, стоящее справа от символа «=» запоминает в переменной или элементе массива, стоящем слева. При присваивании происходит преобразование форм представления (типов), если они не совпадают.

Программа — одна или несколько последовательностей связанных команд (инструкций), которые, будучи выполнены компьютером, реализуют определенную функцию или операцию. Примерами таких функций или операций могут быть: математические вычисления, поиск или сортировка списка объектов, сравнение и отбор объектов по какому-либо критерию, кодирование или декодирование данных, перемещение слов или чисел в закодированной форме в память компьютера, вывод результатов для печати или отображения.

Программист (Programmer) — разработчик прикладных или системных программ, использующий системы программирования.

Программное средство — программа, предназначенная для многократного применения на различных объектах разработчика любым способом и снабженная комплектом программных документов.

Программный продукт — набор компьютерных программ, процедур и связанная с ними документация и данные.

Процессор — центральное устройство компьютера. Назначение процессора: управлять работой ЭВМ по заданной программе; выполнять операции обработки информации. Возможности компьютера как универсального исполнителя по работе с информацией определяются системой команд процессора. Эта система команд представляет собой язык машинных команд (ЯМК). Отдельная команда определяет отдельную операцию

(действие) компьютера. В ЯМК существуют команды, по которым выполняются арифметические и логические операции, операции управления последовательностью выполнения команд, операции передачи данных из одних устройств памяти в другие и пр. В состав процессора входят: устройство управления (УУ), арифметико-логические устройство (АЛУ), регистры процессорной памяти.

Процедурно-ориентированный язык программирования (операторный язык программирования; императивный язык программирования) — язык программирования высокого уровня, в основу которого положен принцип описания (последовательности) действий, позволяющий решить поставленную задачу. Обычно процедурно-ориентированные языки задают программы как совокупности процедур или подпрограмм.

Пустой оператор — не производящий никаких действий, обозначается символом «;», который встречается в программе «сам по себе». Пустой оператор используется там, где по синтаксису требуется наличие оператора, но никаких действий производить не нужно.

Результат функции — выходная переменная, которую формирует функция и значение которой используется затем в том месте программы, где она вызывается. Результат, как любая другая переменная, должен быть определенного типа.

Репозиторий проекта — место, где хранится и поддерживается информация, связанная с проектом разработки программного продукта в течение всего его жизненного цикла.

РЕФАЛ (**РЕ**курсивных **Ф**ункций **АЛ**горитмический язык) — один из старейших языков функционального программирования. Впервые был реализован в 1968 г. в России.

Семантика языка программирования — это смысл, который закладывается в каждую конструкцию языка. Семантический анализ — это проверка смысловой правильности конструкции. Например, если в выражении используется переменная, то она должна быть определена ранее по тексту программы, а из этого определения может быть получен ее тип. Исходя из типа переменной, можно говорит о допустимости операции с данной переменной.

Символьное данное (Character) — тип данных, предназначенный для ввода и отображения алфавитной, цифровой и спецсимвольной

информации; типу соответствуют определенные операции над переменными и функции (строчные).

Синтаксис языка программирования — правила составления предложений языка из отдельных слов. Такими предложениями являются операции, операторы, определения функций и переменных. Особенностью синтаксиса является принцип вложенности (рекурсивность) правил построения предложений. Это значит, что элемент синтаксиса языка в своем определении прямо или косвенно в одной из его частей содержит сам себя. Например, в определении оператора цикла телом цикла является оператор, частным случаем которого является все тот же оператор цикла.

Система программирования (Programming System) — программная среда разработки приложений с использованием языка программирования, библиотеки функций, компилятора или интерпретатора ЯП.

Система управления версиями позволяет разработчикам следить за изменениями кода программного продукта в ходе его разработки, а также управлять различными его состояниями.

Системная программа — программа в машинных кодах, которая поставляется с компьютером (или приобретается отдельно) и рассчитана на выполнение узко специализированных операций. Совокупность системных программ обычно называется системным программным обеспечением (system software).

Составной оператор — последовательность операторов, заключенная в операторные скобки ({ }, `begin end`), образует составной оператор (или блок) и входит в охватывающую его конструкцию как одно целое, т. е. становится с точки зрения транслятора одним оператором.

Средство автоматизации разработки программ (CASE-средство) — программное средство, поддерживающее все процессы жизненного цикла программного обеспечения: анализ и формулировку требований, проектирование, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом.

Стандартное машинное слово — машинное слово, размерность которого совпадает с разрядностью процессора. Большинство команд процессора использует для обработки данных стандартное машинное слово.

Структура данных — способ отображения значений в памяти — размер области и порядок ее выделения (который определит характер процедуры адресации-выборки).

Структура данных линейная — порядок следования элементов данных, который имеет линейный характер и соответствует порядку расположения элементов в памяти.

Структурное программирование — метод программирования, базирующийся на использовании процедурного стиля программирования и последовательной декомпозиции алгоритма решения задачи сверху вниз.

Тип данных — форма представления данных, которая характеризуется способом организации данных в памяти; множеством допустимых значений; — набором операций.

Точка входа — адрес переменной или функции во внутреннем представлении модуля, к которым возможно обращение из других модулей.

Транслятор (translator — переводчик) — программа-переводчик. Преобразует текст программы, написанной на одном из языков высокого уровня, в программу, состоящую из машинных команд.

Указатель — переменная, содержимым которой является адрес другой переменной. Соответственно, основная операция для указателя — это косвенное обращение по нему к той переменной, адрес которой он содержит. Указатель, который содержит адрес переменной, *ссылается* на эту переменную или *назначен* на нее; наоборот, переменная, адрес которой содержится в указателе, называется *указуемой* переменной.

Управление — целенаправленное воздействие управляющего объекта на объект управления, осуществляемое для организации функционирования объекта управления по заданной программе.

Управляющие структуры — структурные операторы. Для того чтобы установить порядок выполнения отдельных операторов, в программу вводят структурные операторы. К структурным операторам относятся составной оператор, условный оператор и цикл. Каждый из структурных операторов в свою очередь состоит из элементарных или других структурных операторов. Управляющие структуры — составной оператор, условный оператор и цикл — равноправны в том смысле, что любая из них может входить в состав другой. В программах встречаются

группы операторов, объединенных в составной оператор, который является элементом цикла или условного оператора, и, наоборот, циклы и условные операторы могут входить в составные операторы или другие циклы. Использование управляющих структур позволяет создавать разнообразные и довольно сложные программы.

Условный оператор if используется в программе, когда нужно выполнить одну или другую последовательность действий в зависимости от выполнения некоторого условия.

Файл (File) — именованный организованный набор данных определенного типа и назначения, находящийся под управлением операционной системы.

Файл ASCII (ASCII-File) — файл, содержащий символьную информацию в коде Latin-1 и символьную разметку.

Файл бинарный (Binary File) — файл, содержащий произвольную двоичную информацию (текст с бинарной разметкой, программа, графика, архивный файл).

Файловая система (File management system) — динамически поддерживаемая информационная структура на устройствах прямого доступа (дисках), обеспечивающая функцию управления данными ОС путем связи «имя—адрес».

Фактические параметры — значения формальных параметров, с которыми будет выполняться тело функции при вызове, определяются списком фактических параметров, которые следуют в вызове функции вслед за ее именем, разделенные запятыми и заключенные в скобки. Перед вызовом функции фактические параметры копируются в соответствующие формальные параметры. Таким образом, функция получает на вход данные через фактические параметры, которые она «видит» как соответствующие им формальные.

Фиксированная точка (Fixed) — простейший тип числовых данных, когда число размещено в машинном слове, и диапазон значений зависит только от разрядности слова.

Формальные параметры — после имени функции в круглых скобках идет список формальных параметров, разделенных запятыми. Формальные параметры — это тоже переменные особого рода, что видно из заголовка — их синтаксис в перечислении напоминает определение обычных переменных. Как и все другие переменные они имеют тип и могут быть обычными перемен-

ными и массивами. Формальные параметры содержат входные данные функции, передаваемые при ее вызове, и могут использоваться только самой функцией в процессе ее работы.

Формат данных (Data format) — стандартизованное представление порции данных для хранения, передачи, отображения.

Функциональная пригодность — набор атрибутов, определяющий назначение, номенклатуру, необходимые и достаточные функции программных средств, заданные техническим заданием (ТЗ) заказчика или потенциального пользователя.

Функциональное программирование (ФП) — методология программирования, согласно которой программа рассматривается как суперпозиция функций. Переменные отсутствуют, нет изменяемых объектов. Такой подход делает программирование ближе к математической записи задачи. Программы, как правило, короче и содержат меньшее количество ошибок, чем в императивном (процедурном) стиле.

Функция выполняет некоторое законченное действие и имеет «стандартный интерфейс» с набором параметров, через который она вызывается из других функций

Цикл ПОКА отличается от цикла ДО тем, что проверка условия проводится до выполнения тела цикла, и если при первой проверке условие выхода из цикла выполняется, то тело цикла не выполняется ни разу.

Цикл ДО применяется в том случае, когда необходимо выполнить какие-либо вычисления несколько раз до выполнения некоторого условия. Особенностью этого цикла является то, что он выполняется хотя бы один раз.

Число с плавающей запятой (Float) — числовое данное, размещенное в машинном слове в форме мантиссы и порядка, что позволяет представлять широкий диапазон значений; предполагает наличие встроенной или эмулируемой арифметики (операции с плавающей запятой). Для использования чисел с дробной частью, а также для расширения диапазона используемых числовых данных вводится форма представления вещественных чисел или чисел с плавающей запятой: $X = m \cdot 10^p$, например $25{,}4 = 0{,}254 \cdot 10^2$, где $0{,}1 < m < 1$ — значащая часть числа, приведенная к интервалу 0,1...1, называемая мантиссой, а p — целое число, называемое порядком. Аналогично, если взять основа-

ние степени — 2, то получим: $X = m \cdot 2^p$, где $0{,}5 < m < 1$ — мантисса, а p — двоичный порядок.

Числовое данное (Numerical) — тип данных, предназначенный для вычислений и составления арифметических выражений, которому соответствуют определенные операции и функции (арифметические).

Элемент данных (элементарное данное) — неделимое именованное данное, характеризующееся типом (напр., символьный, числовой, логический и пр.), длиной (в байтах) и обычно рассчитанное на размещение в одном машинном слове соответствующей разрядности. Это минимальная адресуемая (идентифицируемая) часть памяти — единица данных, на которую можно ссылаться при обращении к данным. Ранние языки программирования (Алгол, Фортран) были рассчитаны на обработку элементарных данных или их простейших агрегатов — массивов (матрицы, векторы). С появлением ЯП Кобол появляется возможность представления и обработки агрегатов разнотипных данных (записей).

ЭВМ (Computer) — универсальный комплекс технических средств (электронная вычислительная машина), предназначенный для программированной обработки информации.

Язык запросов (Query Language) — языковое средство поиска и обработки информации в базе данных или информационно-поисковой системе (ИПС).

Язык программирования (Programming Language) — совокупность средств, предназначенных для описания алгоритмов, реализуемых в программах ЭВМ.

Решения некоторых упражнений главы 2 (язык РЕФАЛ)

9. На языке Рефал запишите образцы, которые соответствуют следующим словесным описаниям:

1) выражение, оканчивающееся двумя одинаковыми символами;

2) выражение, которое содержит по крайней мере два одинаковых терма на верхнем уровне структуры;

3) непустое выражение.

Решение:

1) `e.1 s.X s.X;`

2) `e.1 t.X e.2 t.X e.3;`

3) `t.X e.Y` или `e.X t.Y.`

10. Вычислить результаты следующих сопоставлений:

1) `'abbab' : e.1 t.X t.X e.2;`

2) `'ab(b)ab' : e.1 t.X t.X e.2;`

3) `A(B) C D (A (B)) : e.1 e.2 (e.1);`

4) `'160' : 16 e.X.`

Решение:

1) успешно, когда `t.X` принимает значение `b`;

2) сопоставление неуспешно;

3) успешно, когда `e.1` принимает значение `A(B)`, `e4` принимает значение `C D`;

4) сопоставление неуспешно.

11. В рекурсивной арифметике натуральные числа представляются как 0, 0′, 0″, и т. д. Операция сложения задается соотношениями:

$x + 0 = x$;

$x + y' = (x + y)'$.

Определить функцию `<Addrecurs (e.1)e.2>`, реализующую такую операцию сложения, используя символ `'0'` в качестве `0` и `'1'` вместо символа «'».

Решение:

```
Addrecurs {
  (eX)'0' = eX;
  (eX)eY'1'= <Addrecurs (eX) eY>'1'; }
```

12. Определить функцию `<Addbinary (e.1)(e.2)>`, которая складывает двоичные числа `e.1` и `e.2`.

Решение:

```
Addbinary {
  (eX'0')(eY s1) = <Addbinary (eX)(eY)> s1;
  (eX'1')(eY'0') = <Addbinary (eX)(eY)>'1';
  (eX'1')(eY'1') = <Addbinary (<Addbinary (eX)('1')>)(eY)>'0';
  (eX)(eY) = eX eY; }
```

13. Определить функцию `Less` (<) для целых чисел.

Решение:

```
Less {
       (e1) e2, <- e1 e2>: '-'eZ = T;
       (e1) e2 = F;
       eA = <Less <Format eA>>; }

Format {
       '-'s1 e2 = ('-'s1) e2;
       '+'s1 e2 = (s1) e2;
       s1 '-'e2 = (s1) '-'e2;
       s1 '+'e2 = (s1) e2;
       s1 e2 = (s1) e2; }
```

14. Инвертированная пара в последовательности чисел — это такая пара рядом стоящих чисел, в которой первое число больше второго.

1. Найти в заданной числовой последовательности первую инвертированную пару, для которой сумма членов пары превосходит 100.

2. Определить, превосходит ли сумма членов первой инвертированной пары число 100.

Решение:

```
FGT {
   e1 sX sY e2, <Less sY sX>: T, <Less 100 <+ sX sY>>:
                                 T = sX sY;
   e1 = <Prout 'Нет таких пар'>;
    }
FQw {
   e1 sX sY e2, <Less sY sX>: T,
```

```
    <Less 100 <+ sX sY>>: {T = <Prout 'Больше'>;
                           F = <Prout 'Меньше'>;
                           };
    e1 = <Prout 'Нет таких пар'>;
    }
```

15. Функция $N!$ (факториал) может быть вычислена перемножением чисел натурального ряда от 1 до N. Определить РЕФАЛ-функцию, которая использует этот алгоритм.

Решение:

```
Fact {sN = <Loop F(1) I(1) N(sN)>; }

Loop {
    F(sF) I(sN) N(sN) = <* sN sF>;
    F(sF) I(sI) N(sN) =
        <Loop F(<* sI sF>) I(<+ sI 1>) N(sN)>;
}
```

16. Определить функцию `Reverse`, которая переписывает строку в обратном порядке.

Решение:

```
* Рекурсивное определение
Reverse_r {
    s1 e2 = <Reverse e2> s1;
    = ;
}
```

17. Конечное множество символов может быть представлено строками, включающими те и только те символы, которые входят в это множество. На языках Basic, Java и РЕФАЛ определите следующие функции:

а) вычислить пересечение двух множеств;

б) вычислить объединение двух множеств.

Решение:

```
Isect {
    (sX e1)(e2 sX e3) = sX <Isect (e1)(e2 e3)>;
    (sX e1)(e2) = <Isect (e1)(e2)>;
    ()(e2) = ; }
Union {
    (sX e1)(e2 sX e3) = sX <Union (e1)(e2 e3)>;
    (sX e1)(e2) = sX <Union (e1)(e2)>;
    ()(e2) = e2; }
```

18. Задать функцию, которая определяет, является ли входная строка символов палиндромом.

Решение:

Рассмотрим следующее определение:

1) пустая строка является палиндромом;

2) строка из одного символа является палиндромом;

3) если строка начинается и оканчивается одним и тем же символом, то она является палиндромом тогда и только тогда, когда строка, полученная из нее путем удаления начального и конечного символов, является палиндромом;

4) если не выполнено ни одно из вышеперечисленных условий, строка палиндромом не является.

Реализация функции на языке РЕФАЛ:

```
Palindrom { = True;
   s.1 = True;
   s.1 e.X s.1 = <Palindrom e.X>;
   e.1 = False;  }
```

Оглавление

Голицына Ольга Леонидовна
Попов Игорь Иванович

Программирование на языках высокого уровня

Учебное пособие

Редактор *А. В. Волковицкая*
Корректор *А. В. Алешина*
Компьютерная верстка *И. В. Кондратьевой*
Оформление серии *П. Родькина*

Сдано в набор 02.11.2007. Подписано в печать 14.12.2007. Формат 60×90/16.
Печать офсетная. Гарнитура «Таймс». Усл. печ. л. 31,0. Уч.-изд. л. 31,5.
Бумага офсетная. Тираж 3000 экз. Заказ № 7127.

Издательство «ФОРУМ»
101000, Москва — Центр, Колпачный пер., д. 9а
Тел./факс: (495) 625-32-07, 625-52-43
E-mail: mail@forum-books.ru

По вопросам приобретения книг обращайтесь:

Отдел продаж «ИНФРА-М»
127282, Москва, ул. Полярная, д. 31в
Тел.: (495) 363-42-60
Факс: (495) 363-92-12
E-mail: books@infra-m.ru

Центр комплектования библиотек
119019, Москва, ул. Моховая, д. 16
(Российская государственная библиотека, кор. К)
Тел.: (495) 202-93-15

Магазин «Библиосфера» (розничная продажа)
109147, Москва, ул. Марксистская, д. 9
Тел.: (495) 670-52-18, (495) 670-52-19

Отпечатано с предоставленных диапозитивов
в ОАО «Тульская типография». 300600, г. Тула, пр. Ленина, 109.