С,Н.Лукин

Турбо-Паскаль 7.0

самоучитель

для школьников, студентов и начинающих

1999

Оглавление

От автора

Хочу выразить искреннюю признательность Алексею Михайловичу Епанешникову, внимательно прочитавшему рукопись и сделашему по ее содержанию больше сотни замечаний и нашедшему множество ошибок, одну из которых мне не хочется вспоминать.

Также хочу поблагодарить Алексея Яковлевича Архангельского, просмотревшего рукопись и сделавшего существенные предложения по направленности ее содержания.

Предисловие

Сначала о терминологии. Везде в дальнейшем я буду:

- для краткости вместо термина «Турбо-Паскаль» употреблять термин «Паскаль».
- также везде, где это не вызовет путаницы, словом Паскаль я буду называть не только сам язык Паскаль, но и компилятор, и среду разработки программ:
- пользоваться термином «объект» в его общечеловеческом смысле, несмотря на то, что в языках программирования он имеет специальное значение.

0.1. Кому нужна эта книга?

Это самоучитель. То есть написана книга с расчетом на то, что, изучив ее без посторонней помощи, вы сможете без посторонней помощи писать программы на Паскале и выполнять их на компьютере тоже без посторонней помощи. Материал книги я в течение трех лет проверял на учениках-энтузиастах 6-11 классов 345 московской школы. Проверка, как мне кажется, закончилась нормально - вопросов в процессе изучения ребята задавали немного, а программировать научились. Ответы же на их вопросы я включил в книгу.

Если вы хотите научиться программировать, но никогда в жизни не написали ни одной программы и плохо представляете, как компьютер устроен, читайте эту книгу с начала до конца – вы и программировать научитесь, и об устройстве компьютера узнаете все необходимые сведения.

Если вы школьник или студент, первый год изучающий программирование, и вам предстоит сдавать экзамен по программированию, то изучив книгу и выполнив задания, вы вполне можете рассчитывать на отличную оценку. Исключение – студенты, преподаватели которых уже на первом году обучения предпочитают что-нибудь глубокое или специфичное (например, работу с адресами или оптимальные сортировки). Этого в моей книге нет. Здесь только основы. (Но без них и глубину со спецификой не поймешь.)

Если вы опытный программист, но хотите изучить еще и Паскаль, вам эта книга не нужна - изложение рассчитано на начинающих.

Если у вас **под рукой нет компьютера**, не очень огорчайтесь. Книга снабжена достаточным количеством заданий и решений к ним. Сверяя свои программы с приведенными в ответе, вы всегда сможете сказать, на правильном ли вы пути.

Если вы хотите узнать **Паскаль в полном объеме**, то имейте в виду, что данная книга для этого не предназначена. Во-первых, потому, что Паскаль настолько велик, что в полном объеме во всем мире мало кому нужен. Во-вторых, потому, что для полного его изложения понадобились бы тысячи страниц текста. Цель данной книги не в полноте охвата, а в том, чтобы вы научились программировать и использовать основные средства Паскаля.

Книга учит не только писать программы на листе бумаги, но и **выполнять программы на компьютере**. Все, что вам нужно знать заранее, это как включать компьютер и как запускать Паскаль. Все остальное в книге объяснено.

0.2. Почему Паскаль?

Этот вопрос стоит перед новичками, которые не знают, с какого языка начать. Вот краткий обзор возможных вариантов.

Прежде всего, для полезного, занимательного и веселого изучения основных идей программирования существуют специальные учебные языки, рассчитанные на детей и новичков. Это «Кенгуренок», «Пылесосик», Лого. Кстати, Лого - язык достаточно богатый, чтобы программировать на нем и несложные игры. Но, к сожалению, эти языки мало распространены в России и по ним почти нет литературы. Так что вопрос о них отпадает. Перейдем к рассмотрению профессиональных взрослых языков. Сейчас наиболее известны Бэйсик, Паскаль, Си, Ява в их самых разных версиях.

Но Ява применяется в основном только в сети.

Си – высокопрофессиональный язык, в среде программистов он распространен очень широко, но слишком сложен для восприятия новичком и с него лучше не начинать. Если вам очень хочется программировать на Си, рекомендую начать с Бэйсика или Паскаля. После них освоение Си пойдет гораздо легче.

В качестве языков для обучения студентов и школьников наиболее широко используются Паскаль и Бэйсик. Причина в том, что их современные версии (Borland Pascal for Windows, Delphi, Visual Basic и др.) достаточно ши-

роко распространены во всем мире для разработки профессиональных программ, а сами языки, тем не менее, весьма просты и понятны.

При сравнении Паскаля и Бэйсика нужно помнить, что паскалей и бэйсиков на самом деле много (Pascal, TurboPascal, Borland Pascal, Borland Pascal for Windows, Basic, QBasic, Quick Basic, Visual Basic, Visual Basic .NET и пр.). Оба языка постоянно развиваются и каждой версии присущи свои преимущества и недостатки. Нельзя сказать, что вообще Бэйсик лучше Паскаля, или наоборот, иначе бы на свете остался только один из этих языков. Нельзя с уверенностью предсказать, что кто-то из них в будущем победит. Можно только сказать, что пока каждый силен в своей области и более поздняя версия языка всегда богаче и мощнее более ранней версии.

Исходя из этого, я сравню не Паскаль вообще и Бэйсик вообще, а их версии, наиболее распространенные в российских школах и институтах, а именно TurboPascal 7.0 и QBasic. Если сравнивать их, то на Бэйсике хорошо писать маленькие программы (до 25-50 строк). Они получаются короче и проще, чем на Паскале. Но большие программы на таком Бэйсике из-за плохой структурированности языка получаются труднообозримыми. К тому же Паскаль гораздо богаче, быстрее и мощнее (это естественно, так как он по размеру в десятки раз больше, чем Бэйсик). Паскаль строг и придирчив, изящен и красив.

0.3. Какой из Паскалей?

Любая программа, которую вы встретите в этой книге, является правильной программой в версиях Паскаля TurboPascal 5.5, TurboPascal 6.0, TurboPascal 7.0 и Borland Pascal для DOS, а абсолютное большинство программ - и в более ранних версиях. Это естественно, так как основа языка едина для всех версий. Таким образом, вы можете считать эту книгу учебником по программированию на любой из версий Паскаля. Лично я при написании этой книги использовал TurboPascal 7.0.

Теперь – о работе на компьютере. Каждая версия Паскаля предлагает свой способ работы на компьютере (каждая следующая версия - все более удобный и мощный). В широком смысле этот способ называется средой программирования. Как я уже говорил, среды программирования версий TurboPascal 6.0, TurboPascal 7.0 и Borland Pascal для DOS очень близки между собой в тех рамках, которыми я ограничиваюсь в книге. Я буду учить работать именно в этих средах. Все, что я буду говорить, относится к любой из них. Редкие случаи различий я буду полностью оговаривать. Если же на вашем компьютере установлена другая версия, не очень огорчайтесь, так как основные приемы работы в среде Паскаля одинаковы во всех версиях, а большинство остальных приемов - похожи. Различаются лишь подробности, которые нужны не часто.

0.4. Краткое содержание с рекомендациями

Книга состоит из четырех частей и приложения:

Часть I. Необходимые сведения о компьютере и программе. В этой части Паскалю мы не учимся. Она - для начинающих и для тех более опытных, кто имеет пробелы в знаниях основ компьютера и программирования. В этой части вы узнаете следующее:

- что такое компьютер, программа, цикл, ветвление, процедура, и какая от них выгода;
- что такое языки программирования;
- принцип действия компьютера и его устройств: оперативной памяти, принтера, винчестера и других;
- взаимодействие устройств во время работы компьютера;
- принципы кодирования информации в разных устройствах компьютера.

Часть II. Программирование на Паскале – первый уровень. Цель этой части – провести вас от создания самых простых программ до сложных. Здесь вы учитесь программировать на Паскале самым простым способом - на примерах, то есть по принципу “делай, как я”. Вы научитесь создавать небольшие программы, включающие циклы, ветвления, процедуры и использующие графику и звук. Заканчивается часть созданием и заданием на создание солидной программы. Предполагается, что после выполнения этого задания у вас должно возникнуть ощущение всесильности, то есть вы должны почувствовать, что теперь вам по плечу программа любого размера, и что вам может понадобиться в будущем, так это сведения о работе с теми или иными данными.

Часть III. Программирование на Паскале – второй уровень. Цели этой части:

- Снабдить вас этими самыми сведениями. Вы познакомитесь с действиями над массивами, символами, строками, записями, множествами, файлами и другими типами данных. Вы изучите процедуры и функции с параметрами, модули, расширите свои возможности работы с графикой
- Навести строгость и порядок в ваших знаниях о Паскале. Используемый мной способ изложения на примерах - самый легкий для понимания, но не строгий, а это значит, что если вы сделаете грамматическую ошибку в программе, то иногда не будете знать, в чем эта ошибка. Чтобы иметь оружие на этот случай, в данной части

Паскаль излагается более систематически, а самое главное – приводятся в виде синтаксических диаграмм строгие правила записи всех нужных вам конструкций Паскаля.

Часть IV. Работа в Паскале на компьютере. Она описывает работу в среде программирования TurboPascal 7.0. Вы научитесь вводить программу в компьютер, запускать ее на выполнение, отлаживать с использованием отладчика, сохранять и загружать с диска. В полном объеме среда программирования рассмотрена не будет, но все ее средства, необходимые для уверенной работе на компьютере, будут изложены исчерпывающе.

Приложения. Справочный материал. Если вы никогда не вводили текст в компьютер и не редактировали его, что необходимо при вводе программы, то здесь вы приобретете все нужные для этого умения. Вы познакомитесь со структурой файлов и каталогов на диске, что необходимо при сохранении и загрузке программы. Здесь приводится систематизированный список встречающихся в книге процедур и функций, решения к заданиям и солидный предметный указатель.

Часть I. Необходимые сведения о компьютере и программе

Вы можете не читать эту часть, если в общих чертах знаете следующие вещи:

- *Что такое программа, цикл, ветвление, процедура, и какая от них выгода.*
- *Принцип действия компьютера и его устройств: оперативной памяти, принтера, винчестера и других.*
- *Взаимодействие устройств во время работы компьютера.*
- *Принципы кодирования информации в разных устройствах компьютера.*

В этой части мы не будем программировать на Паскале. А будем знакомиться с перечисленными выше вещами, без которых сознательное программирование невозможно.

Глава 1. Первое представление о компьютере и программе

1.1. Что умеет делать компьютер

Все вы видели компьютер - если не рядом с собой, то хотя бы по телевизору. Обычно он и сам напоминает телевизор, к которому присоединили клавиатуру от пишущей машинки. Только телевизор здесь особый и называется **монитором** или **дисплеем**.

Что же умеет делать компьютер:

- Играть с вами в разные игры.
- Выполнять сложные научные расчеты. Например, он может вычислить траекторию полета космического корабля на Марс.
- Служить хранилищем самых разных сведений и одновременно справочной системой, из которой эти сведения легко и быстро добыть. Например, на любом вокзале Москвы можно подойти к висящему на стене монитору и нажав на клавиатуре несколько клавиш, прочитать на экране сведения о том, есть ли билеты на нужный вам поезд, отправляющийся, скажем, через две недели.
- Выполнять вместо человека несложную, но скучную и утомительную вычислительную и печатную работу. Рассмотрим, например, работу банка. Каждый день в любой банк стекаются со всех сторон тысячи денежных сумм, а другие тысячи денежных сумм забираются из банка. Директор банка должен каждый день точно знать, сколько денег находится у него в банке. Для этого каждый день приходится выполнять тысячи сложений и вычитаний, чтобы сложить все деньги, которые пришли в банк, и вычесть из них все деньги, которые ушли из банка. Кроме этого ежедневно приходится печатать тысячи бумаг, в которых проставляются суммы денег, вложенных в банк и взятых из банка. Ото всех этих обязанностей человека освобождает компьютер.
- Управлять различными машинами и аппаратами там, где человек этого делать не может или не хочет. Например, беспилотным космическим кораблем, подлетающим к Сатурну, управляет компьютер, находящийся на борту аппарата. А, скажем, на заводе компьютер управляет роботом, выполняющим скучную работу по навинчиванию гаек на винты.
- Выполнять обязанности советчика. Например, если в компьютер вложить знания по медицине, а затем сообщить ему, какая у больного температура, анализ крови и пр., он может высказать свое мнение, чем болен пациент, и посоветовать лекарства.
- В самое последнее время компьютер все шире используется, как средство связи со всем миром, гораздо лучшее, чем телефон.
- А еще компьютер помогает инженеру конструировать дома, самолеты, машины, позволяет вам рисовать и делать собственные мультфильмы, он сочиняет стихи и музыку, умеет играть в шахматы на уровне гроссмейстера, может разговаривать человеческим голосом и исполнять музыкальные произведения, предсказывает погоду и делает многое другое.

1.2. Что такое компьютер. Первое представление о программе.

Откуда в компьютере умение делать все описанные выше вещи? Нужно сказать, что когда-то компьютеры ничего такого делать не умели. И их приходилось учить. Как учат компьютер? Примерно так же, как учат людей, рассказывая им, как делать то-то и то-то. Пусть, например, вы живете на 17 этаже многоэтажного дома и к вам в гости приехал человек, никогда не бывавший в городе. Предположим, вы хотите научить его спускаться во двор на прогулку. Для этого вы даете ему такую инструкцию, состоящую из шести команд:

1. Выйти из квартиры
2. Подойти к двери лифта
3. Нажать на кнопку
4. Когда дверь откроется, войти
5. Нажать на кнопку с цифрой 1

6. Когда лифт спустится и дверь откроется, выйти во двор

Если ваш гость умеет ходить и нажимать на кнопки, то помня эту инструкцию, он отныне сможет самостоятельно спускаться во двор.

А как же научить сделать что-нибудь не человека, а компьютер? Например, вы хотите, чтобы компьютер нарисовал на экране монитора синюю тележку. Для этого вы даете ему на специальном, понятном для него языке инструкцию примерно такого содержания:

1. Нарисовать в таком-то месте экрана одно колесо.
2. Нарисовать в таком-то месте экрана другое колесо.
3. Нарисовать в таком-то месте экрана корпус тележки.
4. Покрасить корпус в синий цвет.

Если компьютер умеет рисовать колеса, корпуса и красить их, то он поймет эту инструкцию.

О том, что это за специальный язык, мы поговорим позже. Сейчас же мы скажем, что

инструкция для компьютера по выполнению задания, написанная на специальном, предназначенном для него языке, называется *программой*,

если же она написана на обычном русском или другом человеческом языке в расчете на то, чтобы ее понял не компьютер, а человек, то она называется **алгоритмом**. Таким образом, мы только что написали алгоритм из четырех **команд**[1].

Поскольку у многих компьютеров нет ушей-микрофона (а если и есть, то компьютер неважно разбирает устную речь), программу вы ему не рассказываете вслух, а печатаете ее текст на клавиатуре (по-другому говоря - **вводите** с клавиатуры), откуда она тут же сама собой попадает внутрь компьютера. Отныне компьютер по первому вашему приказу сможет эту тележку рисовать.

Программа для рисования тележки очень простая и короткая. Если же вы хотите научить ваш компьютер делать что-нибудь более сложное, например, играть в шашки, то программу для этого должны будете придумать тоже, конечно, очень сложную и длинную. В этой программе будут встречаться команды такого примерно смысла: если противник сходил так-то, ходи так-то; если твоя шашка попала на последнюю горизонталь, обращай ее в дамку; если шашку противника можно брать, то бери и т.д. Как только вы напишете такую программу и введете ее в компьютер, он сразу же сможет играть в шашки, причем ровно настолько хорошо, насколько хороша ваша программа.

Итак, вы должны запомнить, что

для того, чтобы компьютер что-нибудь умел, он должен иметь внутри себя программу этого умения

И наоборот, если компьютер что-нибудь умеет, это значит, что кто-то когда-то придумал программу этого умения и ввел ее в компьютер. Следовательно, если ваш компьютер умеет играть в игру «Quake», это значит, что внутри него находится программа этой игры, которую кто-то туда ввел. Разучится ваш компьютер играть в «Quake» только тогда, когда вы удалите программу этой игры из компьютера (или нечаянно, или чтобы освободить в компьютере место для других программ).

Таким образом, мы можем определить *компьютер*, как устройство, предназначенное для выполнения широкого круга заданий и вообще для обработки самой разной информации по программе.

Именно работа по программе отличает компьютер от простого карманного калькулятора, который работает только по нажатию клавиш. (Правда, существуют так называемые программируемые калькуляторы, которые могут работать по программе, но раз так, то это уже не совсем калькуляторы, а немножко и компьютеры тоже.)

Вернемся к игре в шашки. Вот, например, ваш компьютер в шашки играть умеет. Как теперь научить играть в шашки другие компьютеры? Можно, конечно, в каждый компьютер ввести упомянутую программу с клавиатуры. Но это долго и утомительно, да и опечаток понаделаешь. Есть способы быстро и безошибочно переносить программы с одного компьютера на другой. Самый популярный из них - использование **дискеты** - маленькой круглой покрытой магнитным веществом пластиковой пластинки в пластмассовом или бумажном футляре, при помощи которой программы переносятся с одного компьютера на другой точно так же, как при помощи магнитофонной кассеты с одного магнитофона на другой переносятся песни.

Когда новенький компьютер выходит с завода, он почти ничего не умеет. Покупатель этого компьютера, чтобы научить его тому, что ему нужно, покупает или берет где-нибудь на время дискеты с программами нужных ему умений и переписывает с них эти программы в свой компьютер. Если нужная программа не существует в природе или просто дискету нигде достать не удалось, то программу приходится придумывать самому и вводить с клавиатуры.

Есть еще два пути, при помощи которых программы могут попасть в ваш компьютер:

---

[1] Имейте в виду, что я дал частное определение программы и алгоритма. В общем случае они определяются, как набор правил для получения нужного результата.

- Вы можете купить **компакт-диски** с готовыми программами и, если ваш компьютер снабжен «проигрывателем» компакт-дисков, то вы вставляете в него компакт-диск и вводите программу в компьютер.
- Если ваш компьютер связан с другими компьютерами при помощи так называемого **модема** или другими способами, то вы можете «перекачивать» программы с других компьютеров на ваш по линии связи.

1.3. Как человек общается с компьютером

Как я уже говорил, информацию, которую человек хочет сообщить компьютеру, он обычно вводит с клавиатуры, дискеты, компакт-диска или по линии связи. Есть и другие способы ввода информации, но о них позже.

Если же, наоборот, компьютер хочет сообщить человеку какую-то информацию, то он обычно показывает ее на экране монитора. Такой информацией могут быть числа, слова, тексты, картинки, мультипликация, видео. По желанию человека компьютер может печатать информацию на бумаге при помощи печатающего устройства, которое называется **принтером**. Кроме этого компьютер может исполнять музыкальные мелодии и даже разговаривать, правда не все компьютеры разговаривают разборчиво. Все это он умеет делать, напоминаем, не сам по себе, от рождения, а только по написанным человеком программам.

Что же обычно делает человек, сидя за компьютером? Все зависит от того, что ему от компьютера нужно. А нужно ему в большинстве случаев вот что:

- Переписать программу с дискеты в компьютер или наоборот - с компьютера на дискету. В этом случае человек просто вставляет дискету в компьютер и нажимает на клавиатуре несколько определенных клавиш.
- Ввести придуманную программу в компьютер с клавиатуры. В этом случае человеку достаточно набрать весь текст программы на клавиатуре.
- Выполнить программу, уже имеющуюся внутри компьютера. Конечно, это нужно человеку чаще всего. Внутри компьютера (если только он не вчера куплен) обычно уже имеется очень много программ, поэтому человек сначала выбирает, какая программа ему нужна. Так, если ему нужна программа, показывающая мультфильмы, то он нажатием нескольких клавиш на клавиатуре выбирает именно ее и приказывает компьютеру ее выполнить (другими словами - **запускает программу** на выполнение). Компьютер выполняет программу, послушно делая все то, что в программе приказано, в результате чего на экране появляется мультик.
- Отвечать на вопросы компьютера. Это происходит только тогда, когда в программе содержится команда компьютеру задать человеку какой-нибудь вопрос. Как видите, без программы компьютер не только ничего не делает, но даже и вопросов не задает. Обычно программа приказывает задать такой вопрос, без ответа на который компьютер не может дальше выполнять задание, данное ему человеком. Например, если компьютер вычисляет траекторию полета к Марсу, то где-то в начале счета он может задать вам вопрос, который вы увидите на мониторе: "Каков стартовый вес ракеты в тоннах?" В ответ вам нужно набрать на клавиатуре правильное число, скажем 2500. Если же вы с компьютером играете, то вам часто приходится отвечать на вопросы типа "Будете ли вы продолжать игру?" и т.п.
- Очень часто в процессе работы с компьютером вам приходится отдавать ему приказы. Например, если вы играете с компьютером в воздушный бой, то нажатием на одну какую-то клавишу вы приказываете самолету выпустить ракету, нажатием на другую - совершить посадку и т.п.

Будем называть человека, профессия которого состоит главным образом в написании программ, **программистом**, а человека, который в основном сидит за компьютером и пользуется готовыми программами, - **пользователем**.

Глава 2. Программа и программирование

2.1. Список команд. Командный и программный режимы

А теперь подробнее рассмотрим, что такое программа. Чтобы лучше это понять, давайте на время забудем о компьютерах. Предположим, в вашем распоряжении находится не компьютер, а настоящий робот. Робот этот умеет понимать и выполнять команды только из следующего списка и никаких других:

Список команд робота:

ШАГ ВПЕРЕД
НАЛЕВО
НАПРАВО
ВОЗЬМИ ПРЕДМЕТ
ОПУСТИ ПРЕДМЕТ
ПОВТОРИ несколько РАЗ выполнение одной из этих команд
СЛУШАЙ ПРОГРАММУ
ВЫПОЛНЯЙ ПРОГРАММУ

Запомните, что робот не умеет делать ничего, кроме того, что упомянуто в списке его команд.
Пусть ваш робот стоит в коридоре и вам нужно, чтобы он переставил стул в комнате на новое место.

Но в списке команд робота нет такой команды "Переставить стул в комнате".Что же делать? Можно идти рядом с роботом и в нужные моменты времени приказывать ему: ШАГ ВПЕРЕД, ШАГ ВПЕРЕД,..., НАЛЕВО,..., ВОЗЬМИ ПРЕДМЕТ... и так далее. В результате стул будет переставлен. Этот режим управления роботом (как, впрочем, и компьютером) называется **командным режимом**. Однако, совсем не обязательно сопровождать робота на каждом шагу. Пусть вы заранее измерили все необходимые расстояния. Тогда достаточно в тот момент, когда робот находится в исходной позиции, сообщить ему инструкцию по выполнению задания, то есть задать точный порядок его действий, приводящих к перестановке стула, а затем приказать выполнить ее. Конечно, инструкция должна состоять только из команд, которые робот понимает и умеет выполнять. Вы уже знаете, что называется такая инструкция программой. Вот она:

Программа для робота	Пояснения для нас с вами
1. ПОВТОРИ 5 РАЗ ШАГ ВПЕРЕД	Робот идет по коридору до дверей
2. НАЛЕВО	Робот поворачивается лицом к дверям
3. ПОВТОРИ 3 РАЗА ШАГ ВПЕРЕД	Робот подходит к стулу
4. ВОЗЬМИ ПРЕДМЕТ	Робот берет стул
5. НАПРАВО	Робот поворачивается к новому месту стула
6. ШАГ ВПЕРЕД	Робот подносит стул к новому месту
7. ОПУСТИ ПРЕДМЕТ	Робот ставит стул на новое место

Очевидно, работая по этой программе, робот правильно переставит стул.

Итак, если вы решили не сопровождать робота на каждом шагу, а заставить его работать по программе, вы совершаете следующие действия:

Последовательность работы человека с роботом

1. Придумываете программу, что не всегда легко, так как нужно хотя бы знать расположение мебели, количество шагов до дверей и т.п.
2. Подходите к роботу, стоящему в исходном положении, и отдаете ему команду СЛУШАЙ ПРОГРАММУ
3. Сообщаете ему программу
4. Отдаете роботу команду ВЫПОЛНЯЙ ПРОГРАММУ

После этого робот работает по программе, то есть выполняет одну за другой команды, из которых составлена программа, в том порядке, в котором он их услышал, в результате чего задание оказывается выполненым, а вы, пока программа выполняется, можете и отдохнуть, чего не могли позволить себе в командном режиме .

Этот режим управления роботом называется **программным режимом.**

2.2. Что важно знать о программе

Чем хороша программа? Ее великое значение в том, что она заставляет робота делать вещи гораздо более сложные, чем те, которые перечислены в списке его команд. По программе робот делает то, что без программы делать не умеет.

Спрашивается, можно ли написать программу для гораздо более сложной задачи, например для перестановки всей мебели на этаже? Разумеется, можно, только программа для этого будет достаточно длинной.

Что требует от нас программа? Она требует абсолютной точности при ее составлении. Если, например, мы в первой команде самую чуточку ошибемся и скажем ПОВТОРИ 6 РАЗ ШАГ ВПЕРЕД (вместо 5 раз), робот проскочит мимо двери, на второй команде повернется и, добросовестно выполняя программу, на третьей команде проломит стену. Если мы перепутаем местами команды 6 и 7, то робот сначала поставит стул, а потом об него же и споткнется.

Запомните, что после того, как вы отдали роботу команду ВЫПОЛНЯЙ ПРОГРАММУ и робот начал ее выполнять, вы не можете программу изменить, пока он не закончил работу. Даже если вы увидите, что робот по вашей программе делает что-то не то, бесполезно кричать на него, хватать его за руку и т.п. Он на вас не обратит внимания. Максимум, что вы можете, это - подбежать к нему и выключить его. После этого вы должны отвести его в исходное положение и только затем можете сообщить ему измененную программу.

Сообщать роботу команды мы обязаны только теми словами, которые приведены в списке команд, потому что робот понимает только их. Если мы вместо команды ВОЗЬМИ ПРЕДМЕТ дадим команду БЕРИ ПРЕДМЕТ, робот нас не поймет и команду не выполнит.

Задание 1: Напишите программу, по которой робот сходит в комнату за стулом и вернется с ним в коридор в исходное положение.

2.3. Понятие о процедуре. Может ли робот поумнеть?

Умен ли наш робот? Судя по его реакции на ошибки в программе, весьма глуп. Умный робот не стал бы проламывать стенку. Однако, наш робот не виноват в своей глупости. Ведь его умственные возможности исчерпываются списком его команд. А список этот очень бедный. Чем он беден?

*Первое***.** В этом списке нет сложных команд, таких как "Наведи порядок в комнате", "Перенеси мебель к другой стенке" и даже такой сравнительно простой, как "Переставь стул". Можете ли вы дополнить список команд робота нужными вам командами? Можете, робот для этого приспособлен. Пусть вы хотите, чтобы робот выполнял команду на перестановку стула. Для этого вы придумываете программу перестановки стула (мы ее уже придумали в п.2.1), затем придумываете, как будет звучать сама новая команда, например ПЕРЕСТАВЬ СТУЛ, и наконец сообщаете роботу программу и говорите ему, что отныне он должен ее выполнять по команде ПЕРЕСТАВЬ СТУЛ. Такая программа называется **процедурой**, а новая команда ПЕРЕСТАВЬ СТУЛ – **обращением к процедуре** или **вызовом процедуры**.

Итак, мы дополнили список команд робота новой командой. Можно ли считать, что робот поумнел? Конечно. Но не очень. И вот почему. Пусть стул находится от дверей не в трех шагах, а в двух. Тогда наш робот, выполняя процедуру ПЕРЕСТАВЬ СТУЛ, как она написана в п.2.1, споткнется об него, а это, конечно, не говорит о его уме. Чтобы переставить стул, где бы он ни был в комнате, робот должен сначала его найти, но команды на поиск нет в списке его команд, а составить из команд этого списка процедуру поиска невозможно. Мы начинаем видеть, чем еще беден список команд робота:

*Второе***.** Он беден не только количеством команд, но и их содержанием. Все его команды касаются только ходьбы и переноса предметов. Фактически, наш робот очень мало умеет, он может только бездумно расхаживать и таскать с места на место мебель. Он не может включить телевизор, так как не имеет хотя бы команды нажатия

на кнопку. Он не может сходить в магазин, так как не умеет считать деньги. Он не может поздороваться с вами, так как не имеет команды что-нибудь произнести. Можете ли вы что-нибудь с этим поделать? Ничего не можете, потому что таким его сделали на заводе. Вам нужен робот с гораздо более разнообразными командами. Кроме этого, хорошо было бы иметь команды для очень мелких движений, например "согнуть средний палец на правой руке". Тогда из таких команд можно было бы составить процедуры вроде "пожми руку" или "нажми кнопку" или "включи телевизор". Но воображаемый робот с такими умениями был бы очень сложным и очень дорого бы стоил.

Вопрос: Что сделает наш робот, получив, находясь в исходном положении, такую бессмысленную программу:

Программа для робота	Пояснения для нас с вами
1.ПЕРЕСТАВЬ СТУЛ	Обращение к процедуре ПЕРЕСТАВЬ СТУЛ
2.НАЛЕВО	Робот поворачивается налево
3.ПЕРЕСТАВЬ СТУЛ	Обращение к процедуре ПЕРЕСТАВЬ СТУЛ

Ответ: Выполняя первую команду (обращение к процедуре ПЕРЕСТАВЬ СТУЛ), робот благополучно переставит стул. Выполняя вторую команду, он повернется лицом к дальней стенке. Выполняя третью команду (обращение к процедуре ПЕРЕСТАВЬ СТУЛ), он на первой же команде этой процедуры (ПОВТОРИ 5 РАЗ ШАГ ВПЕРЕД) врежется в дальнюю стенку.

2.4. Программа для компьютера на машинном языке

Теперь, когда вы понимаете, какую важную роль играет список команд, которые может выполнять робот, настало время вернуться обратно к компьютерам. Программа для компьютера тоже состоит из отдельных команд. Я уже говорил, что человек, который пишет программу для компьютера, называется программистом. Естественно, когда программист пишет программу, ему совершенно необходимо знать список команд, которые может выполнять компьютер. Мы еще поговорим подробнее об этом списке. Но сначала подумаем, а что вообще может компьютер. Вспомним все его умения, перечисленные в 1.1 и позже. Большинство из них сводится в конце концов к тому, что компьютер что-то изображает на экране монитора (числа, тексты, картинки, мультики) или же исполняет какую-нибудь музыкальную мелодию или обменивается информацией с дисками. Программы для всех этих умений состоят из команд компьютера.

Взглянем на список команд новенького компьютера, только что покинувшего заводские стены. Внутри него нет почти никаких программ, поэтому умеет он выполнять только команды, заложенные в него на заводе. Каждая из этих команд заставляет компьютер выполнить какое-то одно простейшее очень маленькое действие, по своей незначительности подобное сгибанию пальца у робота. Отдавать эти команды компьютеру нужно на специальном языке, понятном компьютеру - **машинном языке**. Поскольку изучение машинного языка нам сейчас не нужно, я приведу только смысл некоторых задач, выполняемых командами машинного языка (на русском языке).

Примеры задач, выполняемых командами машинного языка:

Сложить два числа.
Определить, какое из двух чисел больше.
Следующие задачи уже слишком трудны для одной команды машинного языка и под силу только совокупности таких команд:
Изобразить на экране в заданном месте светящуюся точку заданного цвета.
Изобразить на экране заданную букву или цифру.
Включить звук заданной высоты.
Выключить звук.
Запомнить, какую клавишу нажал человек на клавиатуре.

В машинном языке еще много команд, и все они такие же "мелкие". Спрашивается, как же при помощи таких слабеньких команд заставить компьютер сделать хоть что-нибудь путное, скажем, написать слово "ЭВМ" или нарисовать кружочек? Я думаю, вы уже догадались, что нужно сделать - нужно написать программу и сделать ее процедурой. Вот, например, алгоритм программы, изображающей на экране слово "ЭВМ":

1. Изобразить на экране букву "Э"
2. Изобразить на экране букву "В"
3. Изобразить на экране букву "М"

А вот алгоритм программы, вычисляющей выражение (5-7)/(10+40):

1. Вычти 7 из 5
2. Прибавь 40 к 10
3. Раздели первый результат на второй
4. Покажи результат деления на экране монитора

Это ничего, что результат получился отрицательный и дробный. Компьютеры непринужденно справляются с такими числами.

А как же нарисовать кружочек, если компьютер может нарисовать только точку? Если вы посмотрите на экран монитора в увеличительное стекло, то заметите, что изображение любого предмета состоит из маленьких светящихся точек (**пикселов**), которые расположены так близко друг к другу, что сливаются в единое изображение. Примерно то же самое вы видите на фотографии в газете. Вполне можно написать программу, которая рисует рядышком одну за другой множество точек так, чтобы они образовали окружность. Рисунок, поясняющий принцип получения изображения на экране, приведен в 3.4.

2.5. Языки программирования

В чем недостаток команд машинного языка? В том, что действия, вызываемые этими командами, очень мелки. Поэтому программа выполнения даже очень простого задания будет состоять из большого числа команд. Это все равно, что строить дом не из кирпичей, а из косточек домино, - построить можно, но слишком долго и утомительно (зато орнамент из кирпичей на этом доме получится плохой, грубый, из косточек домино – гораздо богаче и подробнее).

Поскольку этот недостаток машинного языка был давным-давно понятен всем программистам, то они составили из команд машинного языка процедуры[2] для выполнения наиболее популярных маленьких заданий, таких как:

- Нарисовать кружочек заданного размера в заданном месте экрана
- Нарисовать прямоугольник заданного размера и формы в заданном месте экрана
- Нарисовать отрезок прямой
- Покрасить заданным цветом определенную область экрана
- Воспроизвести мелодию по заданным нотам
- Написать на экране заданное слово, заданный текст
- Запомнить слово или текст, введенные с клавиатуры
- Вычислить математическую формулу

Как видите, действия, вызываемые этими процедурами, гораздо более крупные, чем у команд машинного языка. Поэтому эти процедуры более удобны для написания программ, хотя бы для таких, как программа, рисующая синюю тележку с надписью "Игрушки". Для ее написания достаточно согласиться с тем, что колесо - это кружочек, а корпус - прямоугольник.

Конечно, хотелось бы иметь все подобные процедуры внутри компьютера. Поэтому давным-давно существуют дискеты и компакт-диски, на которых записаны целые "сборники" таких процедур. И каждый желающий может взять дискету, переписать ее содержимое в компьютер и пользоваться им.

Процедуры на такой дискете записаны не разобщенно, а в комплексе, как составные части особой большой программы. Если мы перепишем эту большую программу в компьютер и запустим ее на выполнение, то она позволит человеку, во-первых, писать собственные программы из упомянутых процедур, а во-вторых, сделает этот процесс удобным, то есть будет обнаруживать многие ошибки в ваших программах, позволит быстро запускать их на выполнение, исправлять, переписывать на дискету и т.д.

Называют такую комплексную программу сложно и по-разному, например, "Среда и компилятор языка программирования высокого уровня". Основное для нас в этом названии - понятие "язык программирования" или будем говорить проще - "язык". Но если язык, то какой? У людей есть русский, английский, китайский языки. Что такое любой из этих языков общения людей? Грубо говоря, это набор букв, слов, знаков препинания и правил, по которым все эти элементы нужно выстроить в цепочку, чтобы получить правильное предложение. Язык программирования – примерно то же самое. Важнейшая часть языка программирования – набор правил, по которым различные объекты (в том числе и обращения к упомянутым процедурам) нужно выстроить в цепочку, чтобы получить правильную программу. Строго говоря, процедуры не являются составной частью языка, однако, вы должны знать, что держа в руках дискету или компакт-диск с надписью “Turbo Pascal” или «C++» или какой-либо другой язык, вы держите в руках целый комплекс программ, который содержит и большое количество этих самых процедур и средства для удобной разработки ваших программ с их использованием.

Языков программирования, как и человеческих языков, придумано много. Зачем? Причина - в разнообразии потребностей программистов, в разных уровнях их квалификации и во многом другом. Так, начинающим вряд ли стоит предлагать Ассемблер, а профессионалу не нужен Лого. Часто разные языки ориентированы на разные предметные области. Например, язык Пролог позволяет удобно описывать логические взаимосвязи в окружающем нас мире, Лого позволяет удобно рисовать фигуры и снабжен для этого соответствующим набором процедур, а вот решать сложные математические задачи с его помощью лучше и не пытаться.

Программистам пока еще не удалось создать язык, удовлетворяющий всех, да и неизвестно, возможно ли вообще его создать, и надо ли.

---

[2] Конечно же, не только процедуры, но и функции, о которых мы будем говорить еще не скоро (13.2). А в языке Си вообще нет процедур, а только функции. Сейчас, когда вы еще практически ничего не знаете о программировании, я вынужден прибегать к упрощениям, чтобы вы не утонули в обилии преждевременных подробностей.

Вот некоторые наиболее популярные языки программирования:

Лого	Logo	язык, рассчитанный на детей, позволяющий просто и занимательно рисовать картинки и программировать простейшие игры
Бэйсик	Basic	язык как для начинающих, так и для профессиональных программистов
Паскаль	Pascal	универсальный язык, позволяющий прекрасно программировать самые разные задачи
Си	C	сложный, мощный язык для профессиональных программистов
Ассемблер	Assembler	сложный, мощный язык, с очень мелкими командами, близкими к командам машинного языка
Лисп, Пролог	LISP, Prolog	языки для создания искусственного интеллекта, роботов

Во всех человеческих языках есть слова «ходить», «есть», «спать», обозначающие понятия, общие для всех языков. Точно так же большинство языков программирования позволяет выполнять общепринятые процедуры, такие, например, как вывод информации на экран, только записываются обращения к этим процедурам по-разному. Прикажем, например, компьютеру к трем прибавить два и результат показать на экране монитора. Вот как эта процедура вызывается на языке Лого:

покажи 3 + 2

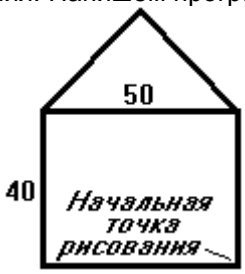
А вот как она вызывается на Паскале:

Write (3+2)

В языках программирования приказы, которые отдают на данном языке, называют не только обращениями к процедурам, но и **командами** (язык Лого и др.), и **операторами** (языки Бэйсик, Паскаль и др.). Между понятиями «обращение к процедуре» и «оператор» существует значительная разница, о которой вы узнаете позже, однако сейчас вам важно знать только одно – команда Лого, обращение к процедуре и оператор являются приказами. Не нужно их путать с командами машинного языка, так как они гораздо «крупнее». Так команда языка Лого *покажи 3 + 2* фактически является обращением к процедуре из нескольких команд машинного языка, которые сначала приказывают компьютеру вычислить сумму, а потом показать ее на экране. Нет команд более мелких, чем команды машинного языка, поэтому любая команда, оператор или процедура на любом другом языке (кроме Ассемблера) сводится в конце концов к выполнению набора команд машинного языка.

2.6. Пример настоящей программы для компьютера на языке Лого

Давайте напишем настоящую программу на настоящем языке программирования. Для этого выберем язык Лого. Он предназначен в основном для рисования. Напишем программу для рисования домика, вот такого:



Начнем с того, что у нас в руках находится дискета с языком Лого. Вставим ее в компьютер. После нескольких нажатий на клавиши посредине экрана возникает вот такая маленькая черепашка:

С этого момента компьютер готов принимать нашу программу и выполнять ее. Занимательность и простота работы с Лого заключается в том, что многие его команды являются командами для черепашки нарисовать на экране те или иные разноцветные линии, что-нибудь покрасить и т.п. Передвигается черепашка по экрану маленькими шагами. Размер экрана по горизонтали и вертикали - несколько сотен шагов.

Из всего длинного списка команд Лого нам для рисования домика понадобятся только две. Приведем примеры их записи с пояснением:

ВПЕРЕД 28 — По этой команде черепашка продвинется вперед на 28 шагов, оставляя за собой тонкий след, то есть фактически она нарисует отрезок прямой длиной в 28 шагов.

НАЛЕВО 60 — По этой команде черепашка повернется на месте налево на 60 градусов.

А теперь напишем программу:

Программа	Пояснения
ВПЕРЕД 40	Черепашка идет вверх и рисует правую стенку дома
НАЛЕВО 90	Собираемся рисовать не крышу, а потолок
ВПЕРЕД 50	Черепашка рисует потолок
НАЛЕВО 90	Собираемся рисовать левую стенку дома
ВПЕРЕД 40	Черепашка рисует левую стенку дома
НАЛЕВО 90	Собираемся рисовать пол
ВПЕРЕД 50	Черепашка рисует пол
НАЛЕВО 90	Готовимся добраться до крыши по правой стене
ВПЕРЕД 40	Черепашка забирается на крышу по правой стене
НАЛЕВО 45	Собираемся рисовать правый скат крыши
ВПЕРЕД 36	Черепашка рисует правый скат крыши
НАЛЕВО 90	Собираемся рисовать левый скат крыши
ВПЕРЕД 36	Черепашка рисует левый скат крыши

Как и программа для нашего воображаемого робота, любая программа для компьютера требует абсолютной точности записи. Нельзя допускать орфографических ошибок - НАЛЕВА, нельзя записывать команды по-другому - ВЛЕВО. Компьютер в этом случае просто откажется выполнять программу. Но если мы, соблюдая эти формальные правила, все же по невнимательности допустим ошибку в программе, компьютер программу выполнять не откажется и, выполнив ее, получит неправильный результат. Например, если в программе пятую сверху команду (для рисования левой стены) мы запишем так - ВПЕРЕД 60 (а не ВПЕРЕД 40), то домик будет выглядеть так:

Так же, как и в случае с роботом, если мы в процессе выполнения программы увидим, что черепашка рисует что-то не то, у нас не будет возможности на ходу исправить программу. Нам или придется ждать, когда она дорисует все до конца или нажатием на клавиши все стереть с экрана и привести черепашку в исходное состояние. После этого программу можно исправлять.

Это я говорил о программном режиме. Лого допускает и командный режим, когда черепашка выполняет команду сразу же, как получит ее с клавиатуры.

Может ли черепашка поумнеть? Да. Объясните черепашке, что составленная программа есть процедура с именем ДОМИК – и отныне вам достаточно будет отдать команду ДОМИК – и черепашка его нарисует.

2.7. Последовательность работы программиста на компьютере

Запишем, в каком порядке проходит работа программиста на компьютере. Она практически копирует порядок работы с воображаемым роботом (см.2.1):

0. Сначала программист получает задачу для компьютера, например, нарисовать домик или вычислить траекторию полета на Марс.

1. Затем он думает, как из команд (операторов), которые ему предоставляет в распоряжение язык программирования, написать программу. На обдумывание и написание программы уходит от нескольких минут до нескольких лет в зависимости от сложности задачи. Почти всегда программист, чтобы не запутаться, пишет сначала алгоритм программы, а потом уже саму программу.

2. Наконец программа написана. Теперь программист включает компьютер и нажатием нескольких клавиш приказывает ему приготовиться к приему программы.

3. Программист набирает всю программу от первой до последней буквы на клавиатуре. При этом программа автоматически по проводу, соединяющему клавиатуру с компьютером, поступает в компьютер и запоминается в его памяти. Программа попала в память компьютера, но это не значит, что компьютер программу “узнал” и “понял”, как узнает и понимает человек. Человек, прочитав какую-нибудь программу, воспринимает ее целиком и хотя бы примерно представляет ее назначение, структуру и т.п. Компьютер же никогда программу у себя в памяти целиком не читает и никогда не понимает ее общего смысла и назначения.

4. Программист нажатием на пару клавиш приказывает компьютеру выполнить программу. Компьютер после некоторой подготовки (куда входит, как главный элемент, перевод программы, написанной на языке программирования, на машинный язык) читает у себя в памяти первую команду программы и выполняет ее, затем читает вторую команду и выполняет, затем третью и т.д. до тех пор, пока не дойдет до конца программы. В результате, если программа составлена правильно, на экране оказывается нарисованным домик или на принтере печатаются результаты расчета траектории полета к Марсу. Но и выполнив правильную программу, компьютер не понял ее смысла и не поумнел.
5. Все программисты допускают в программах ошибки. В результате почти никогда компьютер по только-что написанной программе не делает того, что нужно. Увидев на экране косой домик, программист начинает чесать в затылке и искать в программе команду, виновную в печальном исходе. Найдя ошибочную команду, он исправляет программу и вновь запускает ее на выполнение. Однако результаты снова обычно бывают плачевные. Это происходит потому, что часто программа содержит сразу несколько ошибок. Исправив очередную ошибку, программист снова запускает программу и т.д. Этот захватывающий процесс называется отладкой. Отладка заканчивается, когда программист удовлетворен результатом работы программы (хотя, если программа сложная, это не всегда означает, что в ней ошибок больше нет!).

2.8. Основные приемы программирования

Сведение сложного к простому. Цикл. Итак, чтобы заставить компьютер что-то сделать, нужно написать программу. И тут невольно возникает вопрос - неужели возможно, записывая одна за другой довольно примитивные команды языка программирования, написать программы для всех тех замечательных умений компьютера, некоторые из которых я привел в1.1? Возьмем, например, игру в воздушный бой. Ведь самолетик по экрану должен двигаться! Но в списках команд большинства языков программирования, используемых профессиональными программистами для создания таких игр, нет команды движения. Или возьмем вычисление траектории полета к Марсу. Для ее вычисления нужно решать сложнейшие дифференциальные уравнения высшей математики. Но процедуры языков Фортран, Бэйсик, Паскаль, которые используются для этих целей, не могут ничего, что выходит за рамки школьного курса. И непонятно, в конце концов, как научить компьютер разговаривать, если в нашем распоряжении только команда извлечения из компьютера простого звука заданной высоты.

На все эти вопросы ответ один: если вы хорошо разбираетесь в поставленной задаче, то вы обязательно сможете **разложить ее на много маленьких задач**, каждая из которых вполне поддается программированию, после чего из многих получившихся программок можно собрать одну большую программу, которая и решает задачу. Разберем для иллюстрации несколько основных идей и приемов сведения сложного к простому.

Возьмем воздушный бой. Здесь нужно задаться вопросом - а что такое движение? Рассмотрим иллюзию движения, возникающую на экране кинотеатра. Если вы держали в руках кинопленку фильма, изображающего, скажем, движение автомобиля, то должны были обратить внимание, что она состоит из множества неподвижных слайдов (кадров), на каждом следующем из которых автомобиль находится чуть-чуть в другом месте, чем на предыдущем. Показывая эти кадры один за другим с большой скоростью, мы создаем иллюзию движения автомобиля. Точно так же поступают с созданием движения на экране компьютера. Запишем алгоритм движения по экрану слева направо обыкновенного кружочка:

1. Зададим позицию кружочка в левой части экрана. Перейдем к команде 2.
2. Нарисуем кружочек. Перейдем к команде 3.
3. Сотрем его. Перейдем к команде 4.
4. Изменим (в уме компьютера) позицию кружочка на миллиметр правее. Перейдем к команде 5.
5. Перейдем к команде 2.

Каждая из приведенных команд алгоритма легко программируется на большинстве языков. Кстати, любой компьютер, выполнив очередную команду, автоматически переходит к выполнению следующей (так что нам не обязательно было писать, скажем, во второй команде "Перейдем к команде 3."). Однако, если мы захотим, то можем заставить компьютер изменить этот порядок, что мы и сделали в команде 5, благодаря чему компьютер стал многократно выполнять последовательность команд 2-3-4-5. Такая многократно выполняемая последовательность называется **циклом**. Цикл - основное средство заставить компьютер сделать много при помощи короткой программы.

В коротком промежутке времени после выполнения команды 2 кружок будет появляться на экране и на команде 3 исчезать, но этого достаточно, чтобы человеческий глаз его заметил. Благодаря циклу кружок будет мелькать каждый раз в новом месте, а поскольку смена “кадров” будет очень быстрой, нам будет казаться, что происходит плавное движение кружка.

Теперь перейдем к задаче о траектории. Существует наука - вычислительная математика - которая утверждает, что решение многих самых сложных и страшных математических уравнений можно свести к многократному выполнению четырех действий арифметики, и показывает, как это делать. Грубо говоря, вместо решения одного уравнения она предлагает выполнить пять миллионов сложений. Это как раз то, что нужно компьютеру. Пять миллионов сложений он выполнит за секунду.

И наконец о том, как научить компьютер разговаривать. Акустикам хорошо известно, что любой звук человеческого голоса можно заменить наложением многих простых звуков различной высоты. Наложением сложным и в разные моменты времени разным, однако вполне поддающимся программированию.

***Собственные процедуры.*.** Большую помощь в упрощении программирования сложных задач оказывает возможность, предоставляемая программисту большинством языков, составлять собственные процедуры[3] из команд и операторов языка. Составляется процедура точно так же, как в 2.3 для робота составлялась процедура ПЕРЕСТАВЬ СТУЛ. Пусть, например, вы хотите, чтобы компьютер изобразил на мониторе три поезда, каждый из которых состоит из одного паровоза и четырех вагонов. Сначала мы пишем программу для рисования вагона (прямоугольник и четыре кружочка). Сейчас мы не будем обсуждать, как правильно расположить на экране друг относительно друга прямоугольник, кружочки, вагоны и поезда. Вам остается поверить, что делается это довольно легко. Вот алгоритм программы из 5 команд для рисования вагона:

1. Нарисовать прямоугольник.
2. Нарисовать кружочек.
3. Нарисовать кружочек.
4. Нарисовать кружочек.
5. Нарисовать кружочек.

Для того, чтобы нарисовать четыре вагона, вам понадобилась бы программа из 20 команд. Писать их утомительно, поэтому программисты поступают по-другому. Программу рисования вагона они называют процедурой и придумывают ей имя, скажем, ВАГОН и сообщают его компьютеру. Аналогично процедуре ВАГОН они составляют процедуру ПАРОВОЗ (которую я не буду приводить). Из обращений к этим готовым процедурам они составляют процедуру ПОЕЗД, алгоритм которой будет выглядеть так:

1. Выполни процедуру ПАРОВОЗ
2. Выполни процедуру ВАГОН
3. Выполни процедуру ВАГОН
4. Выполни процедуру ВАГОН
5. Выполни процедуру ВАГОН

И наконец, они пишут программу для рисования трех поездов, которая оказывается совсем короткой:

1. Выполни процедуру ПОЕЗД
2. Выполни процедуру ПОЕЗД
3. Выполни процедуру ПОЕЗД

При отсутствии процедур пришлось бы писать программу из нескольких десятков команд. Выигрыш в объеме работы и в понятности записи программы очевиден.

Ветвление (выбор). У начинающего программиста интерес должен вызвать такой вопрос: как компьютер принимает решения, как он выбирает, какое действие из нескольких возможных нужно выполнить в данный момент? Возьмем тот же воздушный бой. Предположим, при нажатии на клавишу R самолет летит вверх, при нажатии на клавишу V - вниз. Как компьютер чувствует нажатие на клавиши, откуда он знает, что нужно делать при нажатии на каждую из них? Естественно, этот выбор делает программа для игры в воздушный бой, сам по себе компьютер ничего выбрать не может. В программе заранее пишутся процедуры для полета вверх и для полета вниз, а выбор между ними делает специальная команда выбора, имеющаяся в каждом языке программирования. Вот алгоритм выбора между полетом вверх и вниз:

1. Определи, нажата ли какая-нибудь клавиша.
2. Если не нажата, то переходи к команде 5.
3. Если нажата клавиша R, то выполняй процедуру ВВЕРХ.
4. Если нажата клавиша V, то выполняй процедуру ВНИЗ.
5. Продолжай полет.

Здесь команды 2,3,4 - команды выбора. В общем случае команда выбора содержит условие, от которого зависит, будет ли выполняться какая-нибудь команда или группа команд. Это условие может быть самым разным: нажата или нет любая клавиша, нажата или нет конкретная клавиша, больше ли одно число другого, правда ли, что с клавиатуры введено такое-то слово и т.д.

Напишем для примера примитивный алгоритм, позволяющий имитировать вежливое общение компьютера с человеком при включении компьютера:

1. Покажи на мониторе текст "Здравствуйте, я - компьютер, а вас как зовут?"
2. Жди ответа с клавиатуры.
3. Если на клавиатуре человек набрал "Петя" или "Вася", то покажи на мониторе текст "Рад встретиться со старым другом!", иначе покажи на мониторе текст "Рад познакомиться!"
4. Покажи на мониторе текст "Чем сегодня будем заниматься - программировать или играть?"
5. Жди ответа с клавиатуры.

[3] и функции (подчеркну еще раз)

6. Если . . . . . .
. . . . . .

Выбор называют ветвлением по аналогии с разветвляющимся деревом (когда мы залезаем на дерево, мы время от времени делаем выбор, по какой из нескольких веток лезть дальше).

На этом мы завершим умозрительное рассмотрение основных идей программирования. Конкретное их воплощение отложим до изучения Паскаля. А сейчас пришла пора посмотреть на внутреннее устройство компьютера.

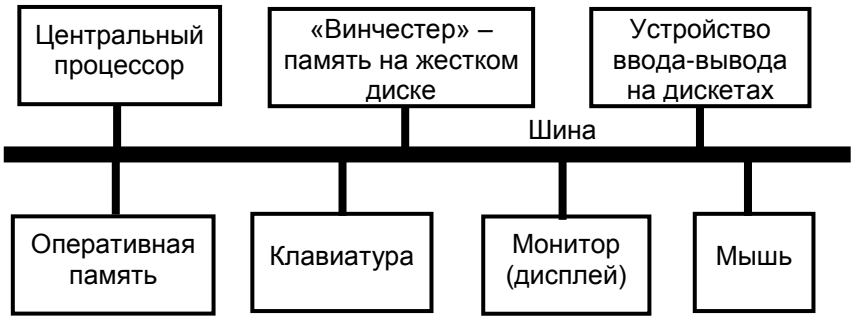
Глава 3. Устройство и работа компьютера

3.1. Как устроен и работает компьютер

Для того, чтобы водить автомобиль, совсем не обязательно знать, как он устроен. Но чтобы быть хорошим водителем, устройство и взаимодействие отдельных частей автомобиля представлять все же необходимо. То же самое можно сказать и о компьютере: можно на нем работать, используя готовые программы, и даже самому программировать, не зная внутреннего устройства компьютера. Однако квалифицированным пользователем и программистом не станешь, не изучив устройство и взаимодействие отдельных его частей.

Компьютеры бывают разные. Вам почти наверняка придется работать на так называемом **персональном компьютере**. Именно про него я сказал в 1.1, что он похож на телевизор с клавиатурой. Вот его обычный внешний вид:

Чтобы легче понять взаимодействие различных частей компьютера, представим себе его устройство схематически:



Схематическое устройство персонального компьютера

На схеме изображено только семь самых необходимых устройств, хотя в компьютере их может быть гораздо больше. Внутри коробки, которая называется системный блок, находятся: центральный процессор (или просто процессор), оперативная память (или просто память), шина, а также почти всегда винчестер и устройство ввода-вывода на дискетах, названное так потому, что с его помощью информацию с дискеты можно в компьютер вводить, а можно и выводить информацию из компьютера на дискету. Кроме того, туда входит много вспомогательных электронных схем и устройств.

Во время работы компьютера все устройства обмениваются между собой информацией. Например, программа с дискеты или клавиатуры отправляется в память. Чтобы информация могла попадать из одного устройства в другое, все устройства соединены между собой общей шиной, которая представляет собой ряд электрических проводников, по которым могут передаваться сигналы.

Рассмотрим, чем занимается каждое устройство компьютера в отдельности. Прежде всего выделим два устройства, образующие "мозг" компьютера - это процессор и оперативная память. Именно эти два устройства осуществляют главную обработку информации, они решают все задачи, вычисляют траектории, обдумывают

шахматные ходы и т.д. Только не надо забывать, что все это они делают, слепо выполняя команды программы, а это значит, что весь интеллект компьютера сосредоточен не в них, а в программе, умен не сам компьютер, а умна находящаяся в нем программа. Поэтому часто вместо того, чтобы сказать "компьютер решил задачу", говорят "программа решила задачу".

Процессор можно назвать "начальником" над остальными устройствами компьютера, так как он во время выполнения программы руководит работой всех устройств. Именно он "понимает смысл" каждой команды программы и выполняет ее сам или приказывает выполнить ее другим устройствам. Однако, процессор почти ничего не помнит. Вернее, его память хороша, но очень мала - он может запомнить всего несколько чисел или букв.

А вот оперативная память специально предназначена для того, чтобы быстро запоминать и быстро вспоминать большие объемы информации, больше ничего она делать не умеет. Короче, процессор и память - как слепой и глухой - по отдельности беспомощны, а вместе вполне способны существовать.

3.1.1. Порядок обмена информацией между устройствами компьютера

Рассмотрим порядок, в котором обычно обмениваются информацией устройства компьютера во время выполнения программы. Пусть мы только-что придумали программу для перемножения двух чисел, одно из которых находится на дискете, а другое должно вводиться с клавиатуры. Вот алгоритм программы:

1. Ввести число с дискеты
2. Ввести число с клавиатуры
3. Перемножить эти числа
4. Показать результат на мониторе

Пусть мы написали эту программу на языке Бэйсик и теперь хотим ее выполнить на компьютере. Для этого мы должны иметь дискету с этим языком. Мы включаем компьютер, он несколько секунд готовится к работе, после чего мы вставляем в устройство ввода-вывода на дискетах (**дисковод**) дискету с Бэйсиком и нажимаем несколько клавиш. Язык с дискеты через шину переписывается (загружается) в память. После этого компьютер сообщает, что готов принимать от нас программу на Бэйсике, и мы эту программу набираем на клавиатуре. (Дискета с Бэйсиком нам больше не нужна и мы вынимаем ее из дисковода.)

Все современные компьютеры могут выполнять программу только тогда, когда она находится в оперативной памяти.

В соответствии с этим требованием наша программа автоматически по мере ввода с клавиатуры отправляется в память и там запоминается. Примерно в это же время мы вставляем в дисковод дискету с одним из двух чисел. Как только вся программа введена, мы нажатием клавиши приказываем компьютеру ее выполнить. И вот что в общих чертах происходит дальше. (Имейте в виду, что следующие два абзаца довольно нудные, в них мы, пожалуй, слишком глубоко забираемся в компьютер. Но без глубины нет вышины.)

Напомним, что на Бэйсике программа состоит из операторов, в нашем случае (допустим для простоты и для соответствия с алгоритмом) - из четырех операторов, хотя на самом деле их будет немного больше. Бэйсик выполняет эти операторы один за другим, по порядку. Вы также знаете, что при выполнении оператор языка программирования заменяется выполнением набора машинных команд. Мы, опять же, чтобы не затемнять изложение подробностями, допустим пока, что в нашей программе каждому оператору соответствует одна команда машинного языка.

Итак, как только программа была запущена на выполнение, процессор прежде всего приказывает памяти послать ему по шине первую команду программы. После того, как эта команда (ввод числа с дискеты) пришла в процессор, он "осознает" ее и отдает приказы устройствам компьютера на ее выполнение в соответствии с тем, как он ее осознал. В нашем случае он отдает приказ дисководу прочесть с дискеты число (пусть это было число 3) и направить его по шине в оперативную память, а оперативной памяти приказывает принять это число и запомнить. Как только число запомнилось, процессор считает команду выполненной и приказывает памяти послать ему вторую команду (которой оказался ввод числа с клавиатуры). Осознав ее, он приказывает компьютеру остановиться и ждать, когда человек введет с клавиатуры какое-нибудь число. Как только человек набрал число на клавиатуре (пусть он набрал -0.25), клавиатура докладывает об этом процессору и тот приказывает ей направить число в память или, возможно, запоминает его сам. После этого он принимает из памяти третью команду (умножение). Внутри процессора имеется арифметическое устройство - своеобразный автоматический карманный калькулятор, способный выполнять четыре действия арифметики. Пусть второе число уже находится в процессоре, тогда он приказывает памяти послать ему по шине первое число, перемножает оба числа, после чего запоминает результат сам или отправляет его в память (предположим, он выбрал память). Наконец, он получает из памяти последнюю команду, согласно которой приказывает памяти же отправить результат (-0.75) на монитор, а тому - принять результат и изобразить его на экране. На этом выполнение программы заканчивается, компьютер останавливается и ждет от человека ввода новой программы или исправления старой.

Итак, мы видим, что работа процессора состоит в том, чтобы считывать из памяти по порядку команды программы, осознавать их, после чего выполнять их самому или приказывать выполнить другим устройствам.

Работа оперативной памяти состоит в том, чтобы хранить программу во время ее выполнения, а также принимать от любых устройств, запоминать и отправлять в любые устройства любую информацию, с которой работает программа. Такая информация, в отличие от программы, называется **данными**. В нашем случае данными являются числа 3 и -0.25 (это **исходные данные** решения задачи), а также -0.75 (это данное является **резуль-**

татом). Программа - это предписание того, что нужно делать с исходными данными, чтобы получить результат, а данные - это информация, над которой производит действия программа и которая зачастую в программе не содержатся. Так. в нашей программе нигде не заданы значения перемножаемых чисел. Оба они находятся совсем в другом месте - одно на дискете, другое вводится с клавиатуры.

Если программа предназначена для сложения 10000 чисел, записанных на дискете, то данными будут эти 10000 чисел. Если программа предназначена для подсчета количества слов в тексте рассказа, вводимого с клавиатуры, то данными будет этот текст. Если программа предназначена для распечатки на принтере изображения с экрана дисплея, то данными будет изображение. Если программа предназначена для распознавания речи, вводимой в компьютер с микрофона, то данными будет звук. В подавляющем большинстве случаев данные во время их обработки хранятся в оперативной памяти.

Взаимодействие различных устройств компьютера можно уподобить взаимодействию нескольких заводов, стоящих вдоль скоростного шоссе (шины) и производящих друг для друга различную продукцию (информацию). При этом память - это не завод, а большой перевалочный склад. А на заводах собственных складов нет или они маленькие. Пусть сегодня один завод произвел для другого большое количество деталей, которое другой завод будет использовать в течение целого месяца. Целиком все детали этот второй завод сегодня забирать не будет, потому что ему складывать их некуда. Первому заводу их тоже негде хранить. Тогда первый завод везет все детали на перевалочный склад, откуда второй завод будет их понемножку забирать по мере надобности.

Назначение других устройств компьютера, кроме процессора и памяти, рассмотрим в 3.4.

3.2. Устройство и размеры оперативной памяти

Представьте себе тетрадный листок в клеточку. В каждую клетку вы имеете право записать карандашом какую-нибудь букву или цифру или знак + или вообще любой символ, который можно найти на клавиатуре. А можете и стереть ластиком и записать другой символ. Много ли букв можно записать на листе? Ровно столько, сколько на нем клеток.

Оперативная память компьютера устроена аналогично этому листу. Только размер ее гораздо меньше, чем у тетрадного листа, а клеточек гораздо больше, и каждая клеточка называется **байтом**. Для запоминания слова КОШКА понадобится 5 байтов. На странице вашего учебника около 1000 букв и других символов (включая запятые, точки и пробелы), значит, для запоминания страницы текста нужно 1000 байтов. Вы можете сами подсчитать, сколько текста может запомнить современный компьютер, если я скажу, что его память редко бывает меньше миллиона байтов. (Если запоминаются не текст, а числа, то место в памяти отводится немножко по другим правилам.)

Оперативная память компьютера электронная. Информация хранится в ней в виде электрических импульсов или потенциалов в миниатюрных электронных схемах и передается из одного места в другое со скоростью близкой к скорости света. Запись, стирание, считывание информации из нее осуществляются по приказам процессора в соответствии с программой меньше чем за десятимиллионную долю секунды.

3.3. Взаимодействие программ в памяти

Этот параграф знать полезно, но пока не обязательно.

Важно помнить, что компьютер работает по программе не только тогда, когда выполняет нашу программу умножения из 3.1, но и до и после этого. Так уж он устроен. Спрашивается, по какой же программе он работает, когда не выполняет нашу. Рассмотрим поподробнее, что происходит в кратко описанный мной в 3.1 период между моментом включения компьютера и моментом начала выполнения нашей программы.

Внутри компьютера в специальном **постоянном запоминающем устройстве** находится программа самопроверки компьютера. Как только вы включаете компьютер, он всегда начинает выполнять именно ее. Если в результате ее выполнения компьютер решит, что его здоровье в порядке, он продолжает работу и обязательно переписывает в память с винчестера (о котором подробнее - позже) основную часть так называемой **операционной системы** (ОС) - комплекса программ, предназначенного для того (скажем пока), чтобы обеспечить человеку и созданным им программам нормальную работу на компьютере. Переписав ОС, компьютер сразу же переходит к ее выполнению и в процессе выполнения останавливается на той ее команде, которая приказывает ему ждать указаний от человека, что ему делать дальше. Вы решаете, например, что вам нужно работать с Бэйсиком, вставляете в дисковод дискету с этим языком и нажатием нескольких клавиш указываете компьютеру запустить Бэйсик в работу. После этого процессор переходит к выполнению следующих команд ОС, которые “осознают” ваше указание и выполняют его, в результате чего переписывают (**загружают**) большую комплексную программу, которой является язык Бэйсик, с дискеты в память и запускают эту программу на выполнение.

Важно понимать, что запуск на выполнение целой программы - Бэйсика явился результатом выполнения очередной команды другой программы - ОС (говорят - ОС **вызывает** Бэйсик или **управление передается** Бэйсику). От того, что начал выполняться Бэйсик, ОС не ушла в небытие, она осталась в памяти, притаилась и ждет, когда Бэйсик, как и положено каждой порядочной программе, решив поставленные человеком задачи, закончит свою работу и уступит место. В этот момент ОС как ни в чем не бывало продолжит свою работу с команды, кото-

рая следует сразу же за той, что запускала Бэйсик (говорят - **управление возвращается** к ОС). Выполнив несколько следующих своих команд и поделав маленькие свои дела, ОС снова натыкается на свою команду, которая приказывает компьютеру ждать указаний от человека, что ему делать дальше. На этот раз человек может пожелать поиграть в какую-нибудь игру. ОС переписывает с дискеты или с винчестера в память и затем вызывает программу этой игры. После окончания игры управление снова возвращается ОС и т.д. Так и проходит с утра до вечера работа на компьютере: после выполнения очередного желания человека ОС получает управление, выполняет некоторую подготовительную работу (чистит память и т.п.) и снова ждет от человека новых пожеланий. Операционные системы бывают разные, самые популярные на персональных компьютерах - MS-DOS и Windows.

А теперь рассмотрим подробнее период между запуском программы-Бэйсика и ее завершением. Получив управление, Бэйсик выполняет некоторые подготовительные действия и останавливается на той своей команде, которая ожидает ввода программы. Вы вводите с клавиатуры свою программу умножения, после чего Бэйсик продолжает работу и следующие его команды отправляют вашу программу с клавиатуры в память. Затем Бэйсик останавливается на другой своей команде, ждущей пожеланий человека. Здесь вы можете пожелать исправлять программу, запустить ее на выполнение, сохранить ее на диске и т.д. Предположим, вы приказываете выполнять программу. Тогда следующие команды Бэйсика, проанализировав ваш приказ, выполняют вашу программу, то есть происходит примерно то, что я подробно описал в 3.1.

Обратите внимание на то, сколько программ находится в этот момент в оперативной памяти. Во-первых, это ОС, которая ждет, когда вам надоест работать на Бэйсике. Во-вторых, это Бэйсик, который выполняет вашу программу, а выполнив, будет ждать от вас дальнейших приказов. И в-третьих, это сама ваша программа умножения. Это обычная практика работы всех компьютеров: в памяти может одновременно находиться от нескольких программ до нескольких десятков. Во многих из них есть команды, которые передают управление другим программам, а затем получают его обратно. Такая передача управления происходит очень часто и зачастую автоматически, без ведома человека. Начинающий программист может ничего этого и не знать. Ему достаточно знать те несколько клавиш, которые он должен нажать, и приказов, которые он должен отдать, чтобы добраться до своего языка программирования и производить там элементарные действия - ввод программы, ее исправление, запуск и т.п.

3.4. Внешние устройства компьютера

Как я уже говорил, процессор и оперативная память образуют "мозг" компьютера. Остальные устройства по отношению к ним являются **внешними** или **периферийными**. Мы рассмотрим назначение самых популярных из них, для чего не очень четко разобьем их на три класса:

- **Устройства ввода** в компьютер информации, поступающей от человека, других компьютеров и аппаратов.
- **Устройства вывода** из компьютера информации, предназначенной для человека, других компьютеров и аппаратов.
- **Внешняя память**, то есть устройства памяти, дополняющие оперативную память.

Разбиение получилось не очень четким потому, что некоторые устройства можно отнести сразу к нескольким классам.

3.4.1. Устройства ввода

Первые три устройства предназначены для ввода информации в компьютер непосредственно от пальцев человека.

1. Клавиатура. Она очень похожа на клавиатуру от обычной пишущей машинки. Предназначена для ввода любой текстовой и числовой информации. На ней есть клавиши для всех английских букв, для всех русских букв, для всех цифр, для других символов, а также клавиши для управления работой компьютера.

2. Мышь. Если вы играете с компьютером в "Зайца и волка" и управляете движением зайца на экране с помощью клавиатуры, у вас не слишком много шансов убежать от волка. И вот почему. Для управления движением объекта на экране используются обычно четыре клавиши: "вверх", "вниз", "налево", "направо". А если спасительный куст находится от зайца где-то наискосок, то вам придется сначала несколько раз нажать на клавишу, скажем, "вверх", а затем - несколько раз на клавишу, скажем, "налево". Пока вы нажимаете, волк до вас доберется.

Хорошо бы иметь возможность перемещать зайца по экрану в любом направлении и с любой скоростью. Эту возможность предоставляет мышь. Вы можете передвигать мышь по гладкому столу в любом направлении, с любой доступной вам скоростью, а заяц на экране будет в точности повторять движения мыши на столе.

Мышь чрезвычайно популярна не только в играх, но и в серьезных программах, так как позволяет быстро перемещать в любое место экрана маленькую фигурку-указатель, называемую **курсором**.

3. Джойстик. Предназначен примерно для того же, что и мышь. Обычно представляет собой коробочку с торчащим из нее рычагом. Коробочка стоит на месте, а рычаг вы можете наклонять в любую сторону, управляя движением объекта на экране.

4. Сканер. Если вы литературовед, то вас вполне могут интересовать вопросы такого типа: "Сколько раз встречается имя "Наполеон" в романе Льва Толстого "Война и мир"?". Поскольку сам роман очень большой, то хотелось бы ответ на этот вопрос поручить компьютеру. Однако тут возникает трудность: чтобы компьютер мог решить эту задачу, текст романа должен оказаться у него в памяти. Для этого вы должны весь текст набрать на клавиатуре - работа на несколько месяцев. Есть способ быстро ввести печатный текст с листа в компьютер. Для

этого используется сканер - прибор, главной составной частью которого является специальное считывающее устройство наподобие телекамеры, передающее изображение листа бумаги с текстом в компьютер.

Итак, изображение текста находится в памяти компьютера. Однако, это не значит, что компьютер "знает", что записано у него в памяти. Ведь компьютер сам не умеет по изображению буквы различить, что это за буква. Нужна специальная программа, которая различает между собой печатные буквы различных шрифтов.

Вы спросите, а как же компьютер различает буквы, вводимые с клавиатуры"? А он различает совсем не буквы, а нажимаемые клавиши.

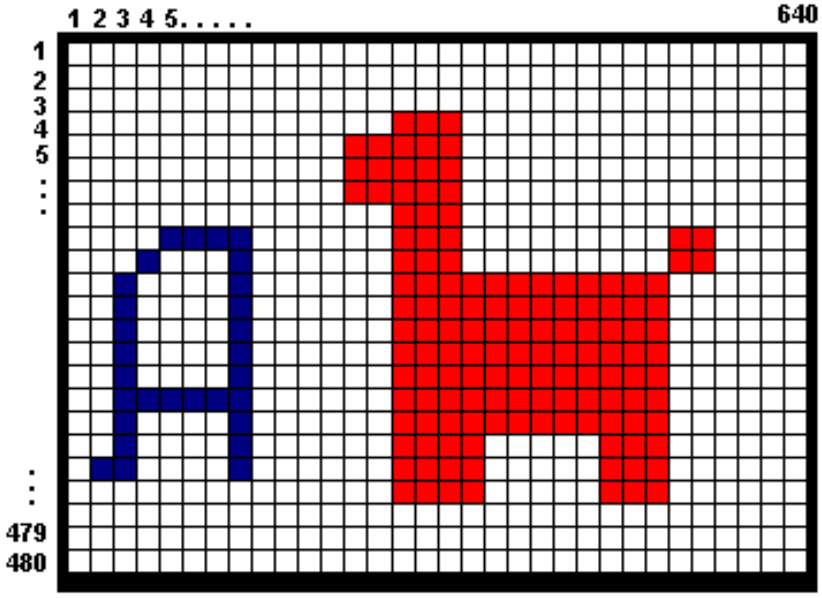
Сканером можно вводить с листа не только текст, но и картинки. Но для распознавания картинок нужны программы еще более сложные, чем для распознавания текста. Так, в мире пока не существует программы, которая бы могла по фотографии отличить собаку от кошки.

5. Микрофон. Предназначен для того, чтобы вы могли что-нибудь сказать компьютеру.

6. Ввод с дискеты. Вы уже знаете, что с дискеты можно вводить (**загружать**) в компьютер программы и другую информацию. Более подробно с дискетами вы познакомитесь чуть ниже.

Устройства вывода

1. Монитор (дисплей). Компьютер не умеет рисовать на экране монитора ничего, кроме светящихся точек (пикселов). Однако, расположив несколько светящихся точек в ряд вплотную друг к другу, получим линию, а сплошь заполнив ими некоторую область экрана, получим изображение фигуры. Пикселы расположены на экране стройными рядами. Каждый пиксел по указанию программы может быть потухшим или гореть заданным цветом (см. рисунок).



Принцип создания изображения на экране монитора

На рисунке вы видите, что на экране умещается 640 столбцов и 480 строк пикселов . Общее количество пикселов получается равным 640x480=307200. Минимальный различимый размер пиксела зависит от качества монитора. Чем больше различимых пикселов умещается на экране, тем размер этих пикселов меньше - и тем тоньше и правдоподобнее рисунки на экране. Общим количеством пикселов управляете вы через специальную электронную схему, находящуюся в компьютере **– видеоадаптер** (**видеокарту**). Если вы заставите видеоадаптер разбить экран на слишком большое количество пикселов, они перестанут быть различимыми и качество изображения не улучшится, а на деле и ухудшится. Количество цветов, которое могут использовать разные видеоадаптеры, тоже разное - от 4 у видеоадаптеров типа CGA до многих тысяч у видеоадаптеров типа SVGA. Вот несколько самых распространенных типов видеоадаптеров, расположенные в порядке возрастания количества пикселов и цветов : CGA, EGA, VGA, SVGA. Последние лет пять в магазинах продают только SVGA.

2.Принтер. Если мы хотим, чтобы числа, текст, рисунки, полученные компьютером, оказались не на экране, а на листе бумаги, мы предусматриваем в программе команду печати их на принтере. Принтер - это, попросту говоря, автоматическая пишущая машинка, подключенная к компьютеру и печатающая по командам программы. Изображение на листе получается из отдельных точек примерно по такому же принципу, как и изображение на экране. По физическому способу получения точек на листе принтеры делятся в основном на матричные, струйные и лазерные. **Матричный принтер** получает черные точки на листе ударами маленьких штырьков (игл) по красящей ленте, которая оставляет след на бумаге. Матричные принтеры самые дешевые и цветными быть не могут. **Струйный принтер** впрыскивает на лист мельчайшие капельки разноцветных чернил из специальных шприцев (сопел), поэтому изображение на листе может быть цветным. **Лазерный принтер** при помощи лазерного луча электризует в нужных точках специальный барабан, после чего тот входит в контакт с красящим порошком. Порошок пристает к барабану только там, где позволяет электрический потенциал, в результате чего на барабане получается изображение. Затем барабан прокатывается по листу бумаги и отдает изображение ему. Качество печати у лазерного принтера самое высокое. Лазерные принтеры самые дорогие, они бывают и цветные.

3.Плоттер - это подсоединенное к компьютеру автоматическое чертежное устройство, которое чертит пером на бумаге чертежи, графики и другие изображения под управлением программы.

4.Звук. Если вы наблюдали, как работает персональный компьютер, то обратили внимание, что во время работы он издает звуки. Это могут быть отдельные редкие попискивания, простенькие мелодии. Их издает устройство, которым снабжаются все компьютеры и которое называется **PC Speaker**. Вы можете купить качественное звуковое устройство, которое называется **звуковая карта**, и тогда вы сможете услышать целый симфонический оркестр и даже внятную человеческую речь. Подробно о программировании мелодий вы прочтете в 8.1.

5.Вывод на дискету. Вы знаете, что на дискеты можно записывать из компьютера программы и другую информацию. Более подробно с дискетами вы познакомитесь чуть ниже.

Внешняя память

Мы рассмотрим три самых распространенных типа внешней памяти.

1. Винчестер (жесткий диск). У оперативной памяти есть два существенных недостатка: **1)** Когда вы вечером выключаете компьютер, все содержимое оперативной памяти стирается. Электронная оперативная память не может что-то помнить, если к ней постоянно не подведен электрический ток. **2)** Оперативная память сравнительно дорога, много ее не купишь, поэтому на большинстве персональных компьютеров сегодня установлено от 128 до 1024 миллионов байтов (сокращенно и приблизительно 1 миллион байтов называют мегабайтом) оперативной памяти. Однако некоторые программы, например, игровые, настолько велики, что требуют для своего запоминания больше 1000 мегабайтов. Это значит, что в компьютеры с маленькой памятью они просто не уместятся, а следовательно, не могут быть выполнены.

Для преодоления этих двух недостатков большинство современных персональных компьютеров снабжаются "**винчестером**" - устройством памяти на жестких магнитных дисках. Запись информации в нем производится на быстро вращающийся диск, покрытый магнитным веществом (см. рисунок). Принцип записи и считывания тот же, что и при записи и воспроизведении песенки на магнитофоне. Цена одного мегабайта памяти винчестера гораздо меньше, чем цена одного мегабайта оперативной памяти, поэтому сегодня на большинстве персональных компьютеров она имеет размер от 40000 до 200000 мегабайтов.

Схема работы памяти на магнитном диске

За дешевизну расплачиваемся быстродействием. Чтобы считать информацию с диска, необходимо подвести магнитную головку к тому месту диска, где записана нужная информация, что занимает сотую долю секунды.

Винчестер используется для хранения самых необходимых и часто используемых больших программ: операционных систем, языков программирования и т.д. Теперь не беда, что большая программа не умещается в оперативной памяти, в ней в каждый момент времени находится только та часть программы, которую необходимо выполнять в данный момент, а остальная часть программы находится на диске и ждет своей очереди, чтобы быть загруженной в память и выполняться.

Кроме этого, вы используете винчестер, выключая вечером компьютер, чтобы переписать на него нужное вам содержимое оперативной памяти, например, создаваемую вами программу, которую вы сегодня только наполовину ввели в память, или полезные вам в будущем результаты сегодняшней работы других программ.

2. Дискета. Часто одним персональным компьютером пользуются по очереди несколько человек. Небрежный пользователь случайным нажатием на клавиши может стереть с винчестера нужную информацию, свою или чужую. В этой ситуациии помогает дискета. Это небольшой гибкий магнитный диск, упакованный в квадратный пластиковый или бумажный конверт. В системном блоке компьютера расположено устройство для считывания и записи информации на дискету - **дисковод**. У дискет есть огромное преимущество перед винчестером - они съемные, а это значит, что важную информацию вы всегда можете переписать с винчестера на дискету, вынуть дискету из дисковода и унести домой, где с нее никто ничего не сотрет. Если с вашей информацией на винчестере что-нибудь случилось, вы приносите из дома дискету, вставляете ее в дисковод и переписываете с нее информацию обратно на винчестер. Вместимость одной дискеты - порядка полутора мегабайтов.

Дискеты имеют еще одно важное применение - с их помощью вы можете переносить понравившиеся вам программы и другую информацию с одного компьютера на другой.

У дискет гораздо ниже быстродействие, чем у винчестера, так как у них гораздо ниже скорость вращения, к тому же обычно после каждого считывания или записи на дискету она прекращает вращаться и для следующего считывания или записи ее приходится раскручивать. Диск же винчестера вращается непрерывно на протяжении всей работы компьютера.

Если учитывать только время считывания или записи информации на дискету, то дискеты менее долговечны, чем винчестер, так как магнитная головка во время работы скользит по поверхности дискеты и постепенно стирает ее магнитный материал, к тому же внутрь дискеты попадает пыль, приводящая к повреждению поверхности дискеты. В винчестере же магнитная головка не соприкасается с поверхностью диска, скользя над ней на воздушной подушке. Пыль внутрь винчестера тоже не попадает, так как он герметически закрыт.

3. Компакт-диски (CD-ROM). Компакт-диски в основном штампуются на заводе, как грампластинки. Информация на них хранится в виде микроскопических бугорков и бороздок под стекловидной поверхностью диска и считывается лазерным лучом.

Компакт-диски сменяемы, надежны, долговечны, вместительны (порядка 600 мегабайтов). На них обычно находятся большие коммерческие программы, изображения, аудиозаписи, видеофильмы для просмотра на компьютере.

Связь компьютеров между собой. Модем. Сети

Для переноса информации с одного компьютера на другой не обязательно использовать дискеты. Если два компьютера расположены рядом на соседних столах, то в простейшем случае их достаточно соединить коротким проводом, и при помощи простенькой программы (которая, кстати, есть в Нортоне) любая информация с винчестера одного компьютера по проводу небыстро переписывается на винчестер другого.

Теперь поговорим о соединении *нескольких* компьютеров. Группу компьютеров, постоянно соединенных друг с другом каким-нибудь способом для обмена информацией, называют **компьютерной сетью**.

Когда эти компьютеры расположены в пределах одного здания, их соединяют специальным кабелем и при помощи мощной операционной системы они могут удобно и с высокой скоростью обмениваться между собой любой информацией. Такая компьютерная сеть называется **локальной**.

А что делать, если компьютеры находятся в разных концах города или земного шара? В этом случае для связи компьютеров используют обычную телефонную сеть. Люди, чтобы переговариваться по телефонной сети, используют телефоны; компьютеры же вместо телефонов используют специальные устройства - **модемы**. Сигналы, посылаемые друг другу компьютерами без модемов, плохо передаются на большие расстояния, модемы же преобразуют эти сигналы в удобный для дальней передачи вид. Такая компьютерная сеть называется **глобальной**. Самый известный пример всемирной глобальной компьютерной сети – **Internet**.

3.5. Кодирование информации в компьютере

Поговорим о том, как физически представлена информация в компьютере. Вот на листе учебника буквы представлены типографской краской, в человеческом мозге информация представлена электрическими импульсами, которые передаются из одной нервной клетки мозга в другую. В компьютере принят тот же способ представления, что и в мозге - из одного устройства компьютера в другое и внутри устройств информация передается электрическими импульсами. Посмотрим поподробнее, как электрические импульсы несут информацию.

Прежде всего заметим, что информация в компьютере - это программы или данные, с которыми эти программы работают.

Из чего состоит программа? Программа на языке программирования состоит из команд, записанных при помощи букв, цифр, знаков математических действий, знаков препинания и других символов[4]. Будем понимать под **символом** любой знак (букву, цифру, знак математического действия, знак препинания и др.), который понимает компьютер. Большинство символов вы можете видеть на клавиатуре.

Из чего состоят данные? Если это числовые или текстовые данные, то они тоже состоят из символов[5]. О графических данных (изображениях) и звуке поговорим чуть ниже.

Таким образом, значительная часть информации в компьютере состоит из символов. Посмотрим, как в компьютере представлены символы. Для этого вспомним, как кодируются символы в азбуке Морзе, активно использовавшейся не так давно для передачи сообщений на расстояние. Каждый символ (буква, цифра) представлен в ней цепочкой точек и тире. Например, буква А представлена, как **.-**, буква Ч - как **---.** . В компьютере каждый символ тоже кодируется, но по-другому – цепочкой из восьми единиц и ноликов. Например, буква А представлена, как 10000000, буква Ч - как 10010111, а цифра 7, как 00110111.

Кстати, вот полезная задачка для будущего программиста: Сколько всего символов можно закодировать цепочкой из восьми единиц и ноликов?

Пока мы с вами говорили о символах и их кодировании безотносительно к тому, какими физическими процессами они представлены в компьютере. Мы были на так называемом «логическом» уровне. Теперь перейдем на физический уровень. Пусть память передает на принтер букву Ч. В этом случае она посылает по шине в течение, скажем, восьми микросекунд, серию из восьми электрических импульсов или промежутков между импульсами:

Первая микросекунда	-	импульс
Вторая микросекунда	-	промежуток
Третья микросекунда	-	промежуток
Четвертая микросекунда	-	импульс
Пятая микросекунда	-	промежуток
Шестая микросекунда	-	импульс
Седьмая микросекунда	-	импульс

[4] Программа на машинном языке состоит не из символов

[5] Опять же, числа в компьютере далеко не всегда состоят из символов-десятичных цифр. Когда компьютер производит над числами арифметические и другие операции, числа представлены совсем по-другому.

Восьмая микросекунда	-	импульс

Как видите, последовательность импульсов и промежутков в серии соответствует последовательности единиц и ноликов в коде буквы Ч. Величина импульса не играет никакой роли, все импульсы в микросхемах компьютера имеют обычно одну и ту же величину, скажем 3 вольта.

Таким же примерно образом обмениваются группами из 8 импульсов все устройства компьютера. В памяти эти группы живут в "замороженном" виде. В каждом байте оперативной памяти или памяти на диске умещается ровно одна такая группа, поэтому говорят, что устройства обмениваются байтами информации.

В оперативной памяти единичка представляется наличием электрического потенциала в определенной точке электронной микросхемы, а нолик - его отсутствием. В памяти на магнитных дисках единичка представляется наличием намагниченности в определенной точке диска, а нолик - его отсутствием или намагниченностью в другом направлении. В компакт-дисках единичка - это бороздка или бугорок в определенной точке диска, а нолик - его отсутствие, то есть участочек с зеркальной поверхностью.

Когда кодируется изображение, то кодируется информация о каждом пикселе изображения (в виде группы единиц и ноликов). Например,

Код 111	-	пиксел горит белым цветом
Код 100	-	пиксел горит синим цветом
Код 010	-	пиксел горит красным цветом
Код 001	-	пиксел горит зеленым цветом
. . . . .		
Код 000	-	пиксел не горит (черный)

Если программа предназначена для распечатки изображения с экрана монитора на цветном принтере, то она просто посылает на принтер по очереди коды информации о каждом пикселе изображения.

При кодировании звука используются разные способы, но факт то, что результатом кодировки являются все те же группы единиц и ноликов.

В заключение отмечу две неточности в моем изложении материала этого пункта. Я говорил, что единички в разных устройствах компьютера представляются наличием потенциала или намагниченности или бороздок и т.д., а вот нолики - их отсутствием. На самом деле в отдельных устройствах может быть и наоборот - единички это отсутствие, а нолики - наличие. Это не принципиально.

Второе: коды чисел в компьютере часто не являются совокупностью кодов цифр, эти числа образующих. Так, число 88 часто не представляется цепочкой 00111000 00111000, а для кодирования чисел используется другой, более экономный способ.

Вывод – любая информация в компьютере закодирована в виде цепочек, состоящих из единиц и нулей, и в таком закодированном виде передается внутри устройств и между устройствами компьютера. Обычно длина цепочки равна 8 и тогда такая цепочка называется **байтом**, а каждый из восьми ноликов или единичек называется **битом**. Таким образом, 1 байт = 8 битов.

_____ – _____

Мне кажется, тех сведений, которые вы получили в этой части, достаточно для того, чтобы приступить к *сознательному* программированию на Паскале.

Часть II. Программирование на Паскале – первый уровень

Цель этой части – научить вас составлять программы, сначала простые, затем все более сложные. А главная цель – добиться у вас ощущения того, что теперь вы можете *самостоятельно* писать программы *любой* сложности. По пути вы узнаете все необходимые для этого средства Паскаля. Причем в этой части я попытался обойтись простыми средствами. Если какое-то средство не казалось мне необходимым для достижения главной цели или было слишком сложным, я его откладывал на следующую часть. Таким образом, если эта часть посвящена простым средствам Паскаля, то следующая – более сложным.

Если вы прочли первую часть, то уже имеете достаточное представление о компьютере и программе, чтобы сознательно программировать на языке Паскаль.

Программа состоит из команд, как дом из кирпичей. Прежде, чем строить дом, нам, конечно, нужно узнать, как выглядят кирпичи. Команды, из которых состоит программа на Паскале и многих других языках, называются **операторами** (см. 2.5). Многие операторы на Паскале являются **обращениями к процедурам** (см. 2.3). Более подробно о смысле этих названий поговорим позже (8.3), а пока не будем делать между ними различия и все подряд будем называть операторами. Каждый новый изученный оператор будет открывать перед нами новые возможности Паскаля, поэтому поставим задачу для начала изучить побольше операторов на примерах их работы в простых программах и только затем перейдем к более сложным программам.

Глава 4. Простые (линейные) программы. Операторы ввода-вывода. Переменные величины

4.1. Процедура вывода Write

Первый оператор, с которым мы познакомимся, занимается исключительно тем, что приказывает компьютеру изобразить на экране монитора ту или иную информацию, состоящую из букв (символов) и чисел. Пусть мы хотим, чтобы компьютер изобразил на экране число 1999. Эту задачу выполняет процедура *Write*. К ней обратимся так - *Write(1999)*. То, что нужно изобразить, положено заключать в круглые скобки. По-английски слово Write означает "пиши". Для тех, кто плохо разбирается в английской транскрипции, я возьму грех на душу и приведу примерное произношение русскими буквами – «райт». Перевод и произношение всех нужных вам слов Паскаля вы найдете в приложении П3 и П4. Сам я настаиваю на умеренно правильном произношении с тех пор, как одна студенточка, указуя перстом на оператор *Write*, мило сказала: «Здесь врите, Сергей Николаевич».

Говорят, что процедура Write **выводит** на экран число 1999, или (по традиции), что процедура Write **печатает** число 1999, хотя, конечно, печатать на экране нельзя. (Правда, Паскаль всегда легко настроить так, чтобы Write выполнялся не на мониторе, а на принтере.)

А теперь поучимся на примерах:

КАК ПИШЕМ ОБРАЩЕНИЕ К ПРОЦЕДУРЕ	***ЧТО ВИДИМ В РЕЗУЛЬТАТЕ НА ЭКРАНЕ***
Write(-500)	**-500**
Write(3 + 2)	**5**
Пояснения: Можно печатать не только числа, но и результаты вычисления арифметических выражений	
Write(3 * 2)	**6**
*Знак * обозначает умножение*	
Write(2 * 2 - 1)	**3**
Write('Хорошо!')	**Хорошо!**
Текст, в отличие от чисел и выражений, нужно брать в одинарные кавычки	
Write('Я изучаю Паскаль')	**Я изучаю Паскаль**
Write(3+2 , 4+4)	**58**

Это не пятьдесят восемь, а два числа: пять и восемь. "К сожалению", они расположены вплотную друг к другу. Чуть ниже вам станет ясно, как преодолеть этот "недостаток" оператора Write.

Write('Это' , 4+4 , 'кошек')	**Это8кошек**

Как видите, один оператор Write может выводить сразу несколько элементов*. Элементы нужно отделять друг от друга запятыми. В этом примере - три элемента: 1)текст 'Это' 2)выражение 4+4 3)текст 'кошек' . Все элементы выводятся в одну строку вплотную друг к другу. Если вся информация, выводимая оператором Write, не умещается в одну строку, то не уместившаяся часть автоматически выводится с начала следующей строки.*

Самое трудное для новичка – не путать запятые и кавычки, находить элементы и отличать текстовые элементы от чисел и выражений. Совет: сначала найдите внутри скобок запятые. Рассмотрим пример:

Write (8, 'котят',3*3,'щенят')	**8котят9щенят**

Здесь запятых три, значит элементов четыре. Вы легко заметите эти элементы, если представите себе, что запятые - это стены, разделяющие элементы.

8 'котят' 3*3 'щенят'

Теперь, чтобы отличить текстовые элементы от прочих, обратим внимание, что они заключены в кавычки.

'котят' 'щенят'

Еще пример:

Write ('Это',4 +4, 'кошек')	**Это8кошек**

Как видите, результат не зависит от количества пробелов (пропусков, пустых мест) снаружи от текстовых элементов, взятых в кавычки. Но пробелы, встретившиеся внутри кавычек, отображаются на экране:

Write('Это',4+4,'ко шек')	**Это8ко шек**
Write('Это ',4+4,' кошек')	**Это 8 кошек**

Write('16+16=' , 16+16) **16+16=32**

Здесь два элемента : текст '16+16=' и выражение 16+16 . Несмотря на то, что текст очень похож на выражение, компьютер узнает его по кавычкам и не вычисляет, а просто воспроизводит, как записано - 16+16= . Любой элемент, заключенный в кавычки, Паскаль считает текстом.

Write(3+2 , ' ' , 4+4) **5 8**

Здесь три элемента. Второй элемент - текст, состоящий из двух пробелов - ' ' . В тексте можно использовать любые символы, имеющиеся на клавиатуре.

Задание 2: Изобразите на листке бумаги в клетку (один символ – одна клетка), что напечатает оператор *Write('12', '5 + 1', 5 + 1, '=', 120+21)*

4.2. Первая программа на Паскале

Конечно, кроме оператора Write в Паскале есть еще много операторов. Но для начального понимания того, как устроена программа на Паскале, достаточно и его. Вот пример программы:

```
BEGIN
   Write('Начали!');
   Write(8+1);
   Write(5-2)
END.
```

Вы видите, что программа состоит из трех операторов Write, записанных друг за другом. Кроме них, программы на Паскале содержат "знаки препинания" - в нашем случае это **служебные слова** BEGIN и END, точка с запятой и точка. Слова BEGIN и END в нашей программе не являются, в отличие от Write, приказами на выполнение каких-либо действий по выполнению задания.

Пока примем такие правила расстановки "знаков препинания":

1. **BEGIN будем ставить в начале программы,** чтобы компьютеру было видно, откуда она начинается. (Это правило мы очень скоро уточним). BEGIN переводится НАЧАЛО, читается "би'гин**".**
2. **END с точкой ставится в конце программы,** чтобы компьютеру было видно, где она кончается. END переводится КОНЕЦ, читается "энд".
3. **Точкой с запятой необходимо отделять операторы друг от друга. Служебные слова BEGIN и END от операторов точкой с запятой отделять не нужно.**

Итак, программа на Паскале, как и на многих языках, состоит из последовательности операторов, выполняемых компьютером в порядке записи. Так, выполняя нашу программу, компьютер сначала выведет на экран слово Начали! , затем число 9, затем число 3. На этом работа программы завершится. На экране мы увидим **Начали!93**

Программу совсем не обязательно записывать в столбик, можно и в строчку, лишь бы было ясно, в какой последовательности записана информация:

```
        BEGIN                    Write(   'Начали!')   ;
   Write( 8  +1          )      ;Write( 5
-2) END.
```

Эта программа записана правильно и, с точки зрения компьютера, не менее изящно, чем предыдущая.

Неважно также, какими буквами - заглавными или строчными, полужирным шрифтом или обычным - записаны названия служебных слов и операторов.

```
bEgin  Wri**Te**( 'Начали!') ;  write( 8**+1**) ;  WRITE( 5-2)   enD.
```

Я для более легкого чтения программ буду выделять полужирным шрифтом некоторые из служебных слов (список всех служебных слов вы найдете в 11.2) и другие общеупотребительные слова Паскаля.

Кроме вышеупомянутых элементов программа на Паскале может содержать **комментарии**. Это - любые пояснительные тексты, взятые в фигурные скобки. Во время выполнения программы компьютер не обращает на них внимания, они ему не нужны, а нужны программисту для более легкого понимания своей программы. Пример:

```
BEGIN
   Write('Начали!');   {Это приказ напечатать слово «Начали!»}
   Write( 8+1) ;
   Write( 5-2)         {А здесь компьютер из 5 вычтет 2 и  напечатает результат}
END.                   {Не забыть бы точку}
```

Не забывайте брать комментарии в фигурные скобки, иначе компьютер подумает, что это операторы, не поймет их и сообщит вам, что у вас в программе ошибка. Правда, есть одно исключение, когда компьютер воспринимает заключенную в фигурные скобки информацию, как директиву на выполнение кое-каких действий, но об этом поговорим позже, в 15.2.

4.3. Выполняем первую программу на компьютере

Если у вас под рукой есть компьютер, то сейчас вам совершенно необходимо вашу программу на компьютере выполнить и посмотреть на результаты. (Если же до компьютера в ближайшем будущем не добраться, смело переходите к следующему параграфу.) Сейчас я вкратце поясню, в каком порядке и какие кнопки нужно нажимать на компьютере, чтобы это сделать. Пояснение рассчитано на тех, кому не терпится поскорее сесть за компьютер и кто уже имеет небольшой опыт работы на нем, в частности умеет вводить в компьютер две-три строчки текста и запускать, например, нужную игру. Остальным же нужно немедленно приступить к изучению части IV, где вас ждет то же пояснение, но уже основательное и подробное, рассчитанное на тех, кто начинает с нуля.

Торопыжкам все равно придется изучать часть IV, но, возможно, попозже. Основательным же скажу, что пока при последовательном изучении части IV вам достаточно остановиться на «Сообщениях об ошибках». «Пошаговый режим» и прочее оставьте на потом.

Итак, вот последовательность действий для опытных по выполнению первой программы на компьютере:

1) Запустите Паскаль. (файл *turbo.exe*). Наверху экрана возникнет меню, а под ним синее (обычно) окно во весь экран с мигающим курсором. Можно вводить программу. Если окно не появилось, то нажмите клавишу *F10*, а затем в меню слово *File* и *New*. (В дальнейшем для краткости я просто напишу **File** → **New**).
2) Введите в это окно программу, как обычный текст в обычном текстовом редакторе.
3) Если получится, сохраните программу на жесткий диск. Для этого **File** → **Save**, а затем в открывшемся диалоговом окне выберите каталог и введите имя файла, в который сохраните программу.
4) Выполните программу. Для этого выполните **Run**→ **Run**, а чтобы увидеть результаты, нажмите *Alt-F5*, что означает: удерживая нажатой клавишу *Alt*, щелкните по клавише *F5*. Выполнив программу в первый раз, поэкспериментируйте – измените содержимое операторов Write – и выполните программу еще раз.
5) Если в вашей программе Паскаль заметил ошибку, он ставит на нее (или недалеко от нее) курсор и сообщает о ней золотыми буквами на красном фоне. Наиболее распространенные для начинающих сообщения об ошибках вы найдете несколькими строками ниже. Более полный список – в части IV. Исправьте ошибку и вернитесь к пункту 3.

Распознаем сообщения компьютера об ошибках

*Итак, вы добились того, чтобы ваша программа печатала нужный результат – **Начали!93**. Теперь давайте поэкспериментируем. Цель эксперимента – научить вас правильно реагировать на сообщения об ошибках, которые выдает Паскаль. Поскольку нужный результат напечатан, то в вашей программе ошибок нет. Придется нам намеренно вводить ошибки и наблюдать за реакцией Паскаля.*

6. Сотрите точку после END. Теперь запустите программу. На экране появится сообщение *Unexpected end of file*, которое переводится *Неожиданный конец файла*. Паскаль нашел эту ошибку в программе и сообщает нам о ней, поставив курсор в строку, содержащую ошибку. Уберите сообщение компьютера клавишей *Esc*.

7. Исправьте эту ошибку и введите другую – сотрите точку с запятой после *Write('Начали!')*. На этот раз сообщение такое – *";" expected*, что значит – *Ждал точку с запятой*. Однако курсор стоит совсем не в том месте, где ошибка, а в начале следующего оператора. Вам придется привыкнуть к тому, что Паскаль не всегда точно определяет место ошибки.

8. Исправьте эту ошибку и введите другую – напишите само имя оператора с ошибкой - *Wite('Начали!')*. Реакция Паскаля – *Unknown identifier*, что означает - *Неизвестное имя*. Имеется в виду имя процедуры *Write*.

9. Исправьте эту ошибку и введите другую - сотрите правую кавычку в операторе *Write('Начали!')*, чтобы получилось *Write('Начали!)*. Реакция Паскаля – *String constant exeeds line*. Перевод вы пока не поймете, но в общем это намек на то, что раз кавычку открыли, то надо ее закрывать.

10. Теперь сотрите левую кавычку. Реакция Паскаля – *Syntax error*, что значит *Синтаксическая ошибка*. Паскаль в затруднении – он знает, где ошибка, но в чем она состоит – не знает.

11. Исправьте ошибки и введите другую - сотрите правую скобку в операторе *Write('Начали!')*, чтобы получилось *Write ('Начали!'* . Реакция Паскаля – *")" expected*, что значит – *Ждал скобку*.

12. Исправьте ошибку и введите другую - сотрите левую скобку в операторе *Write('Начали!')*, чтобы получилось *Write 'Начали!')*. Реакция Паскаля – *";" expected*, что значит – *Ждал точку с запятой*, причем курсор стоит на букве *H*. Вот здесь Паскаль неправ (это не значит, что он глупый, просто невозможно учесть все возможные причины ошибки). Вам придется привыкнуть и к тому, что Паскаль иногда неправильно определяет характер ошибки.

4.4. Процедура вывода WriteLn. Курсор

Оператор WriteLn - читается "райт 'лайн" , переводится - ПИШИ СТРОКУ. Правила его записи и выполнения те же, что и у Write, с одним исключением - после выполнения оператора Write следующий оператор Write или WriteLn печатает свою информацию в той же строке, а после выполнения оператора WriteLn - с начала следующей. Примеры:

ПРОГРАММА	ЧТО ВИДИМ НА ЭКРАНЕ
BEGIN Write ('Ама'); Write ('зонка') **END.**	Амазонка
BEGIN Write ('Ама'); WriteLn('зонка') **END.**	Амазонка
BEGIN WriteLn('Ама'); Write ('зонка') **END.**	Ама зонка
BEGIN WriteLn('Ама'); WriteLn('зонка') **END.**	Ама зонка

Все вышесказанное можно более точно описать с помощью понятия **курсора**. Если вы когда-нибудь вводили текст в компьютер с клавиатуры, то знаете, что вводимый вами текст для вашего удобства автоматически появляется на экране монитора. Вы также, вероятно, обратили внимание на короткую светлую черточку или прямоугольное пятнышко, которое "бежит" на экране перед вводимым текстом. Так, если вы вводите с клавиатуры слово BEGIN, то:

после нажатия на клавишу B на экране вы увидите B_
после нажатия на клавишу E на экране вы увидите BE_
после нажатия на клавишу G на экране вы увидите BEG_ и т.д.

Курсор предназначен для того, чтобы показывать пользователю, где на экране появится следующий символ, который он введет с клавиатуры. Курсор точно так же бежит по экрану впереди текста и тогда, когда информация появляется на мониторе не при вводе с клавиатуры, а во время исполнения программы в результате выполнения операторов Write или WriteLn,

Так вот. Разница между процедурами Write и WriteLn в том, что после выполнения Write курсор остается в той же строке, а после выполнения WriteLn курсор прыгает в начало следующей строки, а значит и все следующие символы волей-неволей будут выводиться в следующей строке.

Оператор WriteLn можно использовать просто для перевода курсора в начало следующей строки. Для этого достаточно написать одно слово WriteLn без скобок.

Задание 3: Определите без компьютера и изобразите на листке бумаги в клетку (один символ – одна клетка), что напечатает программа:

```
BEGIN
   Write (1992);
   WriteLn ( '        Мы начинаем!' );
   WriteLn (6*8);
   WriteLn;
   WriteLn ('Шестью шесть   ' ,          6*6,     '.  Арифметика:' , (6+4)*3)
END.
```

Пояснение: Значок * обозначает умножение. Выполнив задание на бумаге, выполните программу на компьютере и сверьте результаты. Совпадает ли число пробелов между символами? Если не совпадает, еще раз прочтите материал, чтобы понять, почему. Измените число пробелов в разных местах последнего WriteLn. Как изменилась картинка на экране? Почему? Добавте рядом с пустым WriteLn еще один. Что изменилось? Почему?

4.5. Переменные величины. Оператор присваивания

Понятие переменной величины вам известно из школьной математики. Пусть несколько лет назад ваш рост равнялся 130 см. Обозначим этот факт так: $r=130$. Теперь он равен 140 см, то есть $r=140$. Получается, что величина r изменилась.Поэтому она называется переменной величиной. Числа 130 и 140 называются **значениями** переменной величины r.

Любой язык программирования умеет обращаться с переменными величинами. Без них он был бы очень слаб и смог бы извлечь из компьютера только возможности карманного калькулятора. Точно так же алгебра без переменной величины превратилась бы в арифметику. Однако, преимущества применения переменных величин нам откроются позже, а пока наша задача - к ним привыкнуть.

Что же мы можем делать с переменными величинами, программируя на Паскале? Прежде всего, мы можем задавать компьютеру значение той или иной переменной величины. Это мы можем сделать при помощи нового оператора, который называется **оператором присваивания**. Так, если мы хотим сказать, что a имеет значение 6, то должны записать $a := 6$. Как видите, вместо значка **=** употребляется значок $:=$. Он называется **знаком присваивания** , а сама запись $a:=6$ называется оператором присваивания. Говорят, что величине a присваивается значение 6. С момента выполнения оператора $a:=6$ компьютер будет помнить, что a равно шести.

Справа от значка $:=$ в операторе присваивания можно писать не только числа, но и переменные величины, и выражения. Выражение пока будем понимать так, как его понимает школьная математика. Например, после выполнения следующего фрагмента программы: $\ldots\ a:=2*3+4;\ \ b:=a;\ \ y:=a+b+1\ldots$ компьютер будет знать, что a равно 10, b равно 10, y равно 21. Еще несколько примеров:

ФРАГМЕНТ ПРОГРАММЫ	ЧТО ЗАПОМНИТ КОМПЬЮТЕР
v := -2+10; h := 10*v; s := v+h	v=8 h=80 s=88
t := 0; n := 2*t+40; z := -n	t=0 n=40 z= -40

Необходимо помнить, что компьютер выполняет оператор присваивания "в уме", то есть результат его выполнения не отображается на экране. Если мы хотим видеть результат выполнения оператора присваивания, используем WriteLn. Примеры:

ФРАГМЕНТ ПРОГРАММЫ	ЧТО ВИДИМ НА ЭКРАНЕ
v := -2+10; h := 10*v; s := v+h; WriteLn (s)	88
v := -2+10; h := 10*v; s := v+h; WriteLn ('s')	s
v := -2+10; h := 10*v; WriteLn (v+h)	88
v := -2+10; WriteLn (v+10*v)	88
v := -2+10; h := 10*v; s := v+h; WriteLn (v, ' ',h, ' ',s)	8 80 88
v := -2+10; h := 10*v; s := v+h; WriteLn (v+1000, ' ',10*h, ' ',s)	1008 800 88

Нужно иметь в виду, что слева от знака $:=$ может стоять только переменная величина, но не число и не выражение. Можно писать $c:=34$, но нельзя писать $34:=c$. Можно писать $z:=f-v+990$, но нельзя писать $f-v+990:=z$. Правило это принято потому, что оператор присваивания сначала смотрит или вычисляет, какое значение имеет правая часть, а затем присваивает это значение переменной, стоящей в левой части. Нет смысла присваивать значение числу или выражению.

Обратите внимание на один важный момент. Когда школьник видит выражение (например, $d+2d$), он не обязательно его вычисляет. Он может его преобразовать или, скажем, упростить (получив $3d$). Компьютер же, видя выражение, сначала его, может быть, и упростит, но затем обязательно вычислит. А для этого он должен знать численные значения входящих в него величин (в нашем случае это величина d). Таким образом, вычисляя правую часть оператора присваивания (например, $y:=a+b+1$), компьютер должен обязательно заранее знать, чему равны переменные, из которых эта правая часть состоит (в нашем случае это a и b). Ответственность за это знание лежит полностью на программисте. Пусть забывчивый программист записал такой фрагмент: $\ldots\ a:=10;\ y:=a+b+1\ \ldots$, нигде в программе не придав b никакого значения. Естественно, при вычислении выражения $a+b+1$ компьютер не будет знать, чему равно b. В такой ситуации разные языки программирования поступают по-разному. Некоторые просто отказываются вычислять выражения, некоторые подставляют вместо b нуль, Паскаль же иногда может поступить довольно вредно: вместо b он подставит, что бог на душу положит, и получит сногсшибательный результат, например, игрек становится равным -8904 .

Подведем предварительные итоги обсуждения очень важного оператора присваивания:

Оператор присваивания состоит из знака присваивания := , слева от которого пишется переменная, а справа - число, переменная или выражение. При выполнении оператора присваивания компьютер "в уме" (то есть ничего не показывая на мониторе) вычисляет правую часть и присваивает вычисленное значение переменной, стоящей в левой части.

Задание 4: Какое значение будет присвоено переменной **t** после выполнения фрагмента: *k:=1+2; s:=2*k; t:=6-s* ?

4.6. Описания переменных (VAR)

В 4.5 я писал фрагменты программ, содержащих переменные величины. А теперь пришло время писать такие программы целиком. Но для этого я должен написать кое-что выше BEGIN, чего раньше не делал. Вот пример программы:

```
VAR  a,b : Integer;
BEGIN
   a:=100;
   b:=20;
   WriteLn (a + b)
END.
```

Очевидно, эта программа напечатает число 120. В программе новостью для нас является только первая строка. Называется она **описанием** переменных величин a и b. Я пока не буду излагать правила составления описаний переменных. Скажу только, что описание начинается со служебного слова VAR (читается ВАР, это сокращение от английского variable - переменная). После него записываются имена всех переменных, встречающихся в программе с указанием через двоеточие типа значений, которые каждая переменная имеет право принимать. В нашем случае я записал имена переменных a и b, так как только они встречаются между BEGIN и END. Слово Integer (читается " 'интеджер", переводится ЦЕЛЫЙ) указывает, что a и b имеют право быть целыми числами, а не дробными. Смысл подобного ограничения в том, что Паскаль по-разному работает с целыми и дробными числами, а поскольку он язык строгий, то требует, чтобы программист точно указал, к какому из этих двух типов относятся числа.

Зачем нужно описание? Для понимания этого сначала нужно вспомнить, для чего нужна компьютеру память и как она устроена (см.3.2), а также порядок работы человека на компьютере (см.2.7).

После того, как программист ввел программу в память, он приказывает компьютеру ее исполнить. Но компьютер при этом не сразу принимается выполнять программу, а сначала совершает **компиляцию**, то есть перевод программы с Паскаля на собственный машинный язык. (Часто вместо термина «компиляция» употребляют более общий термин «трансляция»). Во время компиляции компьютер производит некоторые подготовительные действия, одним из которых является отведение в памяти места под переменные величины, упомянутые в программе. При этом компьютер "рассуждает" так: Раз в программе упомянута переменная величина, значит она в каждый момент времени будет иметь какое-то значение, которое, хочешь не хочешь, надо помнить. Лучше, чтобы не спутаться, заранее отвести в памяти определенное место для запоминания текущего значения каждой переменной величины и только потом уже выполнять программу. Будем называть место, отведенное в памяти под данную переменную, **ячейкой**.

Теперь нам понятно, зачем в паскалевской программе нужно описание - чтобы перечислить компьютеру переменные, под которые он должен отвести ячейки в памяти. Если мы забудем упомянуть в описании какую-нибудь переменную, то под нее в памяти не будет отведена ячейка и компьютер не сможет ее запомнить, а значит и вообще не сможет с ней работать. Паскаль строг к программисту, он заставляет его самого перечислять в описании все переменные, встречающиеся в программе. Бэйсик более добр - если программист поленился написать описание, Бэйсик сам просматривает программу и выискивает переменные величины, чтобы отвести для них место. Однако, не всегда доброта лучше строгости . . .

Итак, при решении задачи в памяти компьютера находится программа решения задачи, а в другом месте памяти - значения переменных, описанных в программе. Кстати, вы можете узнать, сколько байтов в памяти займет текст программы, если посчитаете, сколько в ней букв и других символов (включая пробелы).

Вы можете сказать: Зачем хранить значения 100 и 20 в каком-то другом месте, если они содержатся прямо в тексте программы, которая и без того хранится в памяти? Получается, что эти числа хранятся в двух разных местах памяти. Ответ на этот вопрос выходит за рамки книги.

4.6.1. Integer и LongInt

Итак, в Паскале принято правило, что если человек описал переменную, как Integer, то он разрешает ей принимать значения только целого числа. Число типа Integer занимает в памяти два байта. Значит, под переменные a и b компьютер отводит в памяти ячейки по два байта каждая. Два байта - это маленький объем памяти и уместиться в него может лишь небольшое целое число, а именно - число в диапазоне от -32768 до 32767. Если бы в предыдущем примере вам понадобилось взять a=40000, то Паскаль получил бы неправильную сумму.

Для того, чтобы переменная имела право принимать значения больших целых чисел, она должна быть описана не как Integer, а как **LongInt** (сокращение от Long Integer - Длинное Целое, читается “лонг’инт”). Под переменную типа LongInt компьютер отводит в памяти 4 байта и она может принимать значения в диапазоне от -2147483648 до 2147483647.

Задание 5: Население Москвы равняется a=9000000 жителей. Население Нью-Васюков равняется b=1000 жителей. Напишите программу, которая определяет разницу в числе жителей между двумя городами. Используйте переменные величины.

Задание 6: Попробуйте ввести ошибку. Опишите переменные величины не как LongInt, а как Integer. Какова реакция Паскаля?

4.7. Что делает оператор присваивания с памятью

Рассмотрим пример программы:

```
VAR  a,b,y : Integer;
BEGIN
    a:= 10;
    b:= 6;
    y:= a+b+1;
    WriteLn (y+200)
END.
```

В программе между BEGIN и END встречаются три переменные, поэтому все они перечислены в описании VAR a,b,y : Integer . Компьютер отведет для них в памяти три двухбайтовые ячейки.

В 4.5 я рассказывал о работе оператора присваивания, используя такие выражения, как "компьютер знает", "компьютер помнит". Но нам необходимо более строгое понимание работы этого оператора, понимание "ближе к железу":

Выполняя оператор присваивания (например, *y:=a+b+1*), компьютер сначала смотрит на его правую часть (*a+b+1*). Если в ней встречаются переменные (в нашем случае это *a* и *b*), то компьютер перед вычислением ищет их значения в отведенных под них ячейках памяти (и находит там 10 и 6), подставляет эти значения в правую часть и вычисляет ее. Затем вычисленное значение (17) компьютер записывает в ячейку памяти, отведенную под переменную, поставленную в левой части (*y*).

Таким образом, когда мы говорим "Компьютер запомнил, что a равно 2", мы подразумеваем "Компьютер записал в ячейку памяти, предназначенную для a, число 2".

А теперь рассмотрим, как будут заполняться информацией ячейки a,b,y в процессе выполнения нашей программы. В самом начале выполнения паскалевской программы в них находится неизвестно что. Первым выполняется оператор $a:=10$. Согласно только что приведенному определению оператора присваивания в ячейку a будет записано число 10. Затем выполняется оператор $b:=6$ и в ячейке b появляется шестерка. Затем выполняется оператор $y:=a+b+1$. Компьютер смотрит, что находится в ячейках a и b, видит там 10 и 6, подставляет их в выражение $a+b+1$, получает 17 и записывает в ячейку y. Наконец выполняется оператор $WriteLn\ (y+200)$. Компьютер заглядывает в ячейку y, видит там 17, вычисляет 17+200 и выводит 217 на экран.

Схематически этот процесс можно изобразить так:

ПОРЯДОК ИСПОЛНЕНИЯ ОПЕРАТОРОВ	ЧТО НАХОДИТСЯ В ЯЧЕЙКАХ ПАМЯТИ			ЧТО ВИДИМ НА ЭКРАНЕ
	a	b	y	
a:= 10	10	?	?	
b:= 6	10	6	?	
y:= a+b+1	10	6	17	
WriteLn (y+200)	10	6	17	217

Теперь мы можем также уточнить работу оператора WriteLn:

Если в скобках оператора WriteLn встречаются выражения с переменными величинами, то Паскаль находит в памяти значения этих величин, подставляет их в выражения, вычисляет выражения и результат выводит на экран.

Задание 7: Поменяйте местами операторы $b:=6$ и $y:=a+b+1$. Что произойдет?

4.7.1. Оператор присваивания меняет значение переменной величины

Пока я не рассматривал программы, в которых переменные меняют свою величину. Теперь настало время такую программу рассмотреть:

```
VAR k : Integer;
BEGIN
   k:=10;  WriteLn (k);   k:=25;   WriteLn (k);   k:=4;   WriteLn (k)
END.
```

Запишем схематически процесс изменения информации в ячейке k:

ПОРЯДОК ИСПОЛНЕНИЯ ОПЕРАТОРОВ	ЧТО НАХОДИТСЯ В ЯЧЕЙКЕ ПАМЯТИ k	ЧТО ВИДИМ НА ЭКРАНЕ
k:=10	10	
WriteLn (k)	10	10
k:=25	25	
WriteLn (k)	25	25
k:=4	4	
WriteLn (k)	4	4

Как видите, в процессе работы программы содержимое ячейки k меняется. Так, при выполнении оператора $k:=25$ там вместо значения 10 появляется 25. А куда же девается десятка? Она стирается, то есть компьютер забывает ее безвозвратно. Здесь действует общий принцип работы всех компьютеров:

Если в какое-нибудь место памяти или диска записывается новая информация, то старая информация, записанная там раньше, автоматически стирается, даже если она кому-то и нужна.

Раз теперь вместо 10 в ячейке k находится 25, то оператор $WriteLn\ (k)$ печатает уже 25. Следующий оператор $k:=4$ запишет на место 25 четверку, а $WriteLn\ (k)$ ее напечатает.

А что напечатает следующая программа?

```
VAR f : Integer;
BEGIN
   f:=30;
   f:=f+4;
   WriteLn (f)
END.
```

Оператор $f:=30$ запишет в ячейку f число 30. А что сделает странный оператор $f:=f+4$? По определению оператора присваивания он сначала вычислит правую часть $f+4$, подставив туда вместо f его значение, взятое из ячейки, и получит 34. Затем число 34 будет записано в ячейку, отведенную под переменную, обозначенную в левой части, то есть опять в ячейку f. При этом старое значение 30 будет стерто.

Таким образом, оператор $f:=f+4$ просто увеличивает число в ячейке f на четверку или, другими словами, увеличивает f на 4.

Задания 8-10:

Определите без компьютера, что будет напечатано при выполнении следующих фрагментов программ:

8) a:=100; a:=10*a+1; WriteLn (a)

9) a:=100; a:=-a; WriteLn (a)

10) a:=10; b:=25; a:=b-a; b:=a-b; WriteLn (a,' ',b)

4.8. Имена переменных

Мы привыкли переменные величины обозначать буквами (a, s, d ...). Большинство языков программирования, в том числе и Паскаль, позволяет обозначать переменные не только буквами, но и целыми словами. Вот два равносильных фрагмента программы:

a:=3;
b:=4-a;
WriteLn (a,b+50)
	Summa:=3;
	ROBBY:=4-Summa;
	WriteLn (Summa,ROBBY+50)

В том и другом случае будут напечатаны числа 3 и 51. Очевидно, компьютеру все равно, как мы обозначаем переменные величины, в смысл имен он не вдумывается и не удивляется, что переменная *Summa* никакой суммой не является, а просто числом 3.

Будем называть обозначение переменной **именем** или **идентификатором** этой переменной.

Правило:

Именем переменной в Паскале может служить любая последовательность цифр, латинских букв и знака подчеркивания, не начинающаяся с цифры.

Примеры правильной записи имен:

a
x
velichina
zzz
polnaja_summa
tri_plus_dva
s25
k1
_k1
___
a1b88qq
oshibka

Примеры неправильной записи имен:

ж	- буква не латинского алфавита
polnaja summa	- содержится символ (пробел), не являющийся буквой, цифрой или знаком подчеркивания
2as	- начинается с цифры
Domby&Son	- содержится символ & , не являющийся буквой, цифрой или знаком подчеркивания

4.9. Математика. Запись арифметических выражений

Если вы - школьник не самых старших классов, то не все, что здесь написано, будет вам понятно. Не огорчайтесь, при дальнейшем чтении непонятные вещи вам не понадобятся.

В правой части оператора присваивания и в операторе WriteLn мы записывали выражения, имеющие численное значение (например, $a+b-8$). Такие выражения называются **арифметическими**. В будущем мы увидим, что выражения могут быть не только арифметическими. А сейчас рассмотрим математические возможности Паскаля.

Четыре действия арифметики (и еще два) обозначаются в Паскале следующим образом:

ДЕЙСТВИЕ	РЕЗУЛЬТАТ	СМЫСЛ
2 **+** 3	5	плюс
4 **-** 1	3	минус
2 ***** 3	6	умножить
10 **/** 5	2	разделить
17 **div** 5	3	целочисленное деление
17 **mod** 5	2	остаток от целочисленного деления

На уроках математики мы привыкли писать $ab+cd$, подразумевая: a умножить на b плюс c умножить на d. В Паскале это выражение мы обязаны писать так: $a*b+c*d$. Иначе компьютер подумает, что нужно к переменной, имеющей имя ab, прибавить переменную, имеющую имя cd. Во избежание двусмысленности знак умножения положено писать всегда. Например, $a * (b+c)$.

Скобки. Ввиду того, что с клавиатуры всю информацию приходится вводить символ за символом в одну строку, ввод двухэтажных выражений, таких как

$$\frac{a+1}{b+1},$$

очень затруднен. Поэтому для обозначения деления и выбрана косая черта. Это выражение на Паскале положено записывать так: $(a+1)/(b+1)$. Если бы мы не поставили скобок, то выражение получилось бы таким $a+1/b+1$, а это неправильно, так как компьютер, как и мы, всегда перед сложением и вычитанием выполняет умножение и деление, поэтому в последнем случае он бы сначала разделил 1 на b, а затем к результату прибавил a и 1.

Вопрос: когда в выражениях можно ставить скобки? Ответ: всегда, когда у вас возникают сомнения в правильной очередности действий. Лишняя пара скобок не помешает. Пример: записать на Паскале выражение:

$$\frac{1+\frac{a}{2+ab}}{3+a}\cdot b$$

Его можно было бы записать так: $(1 + a / (2+a*b)) / (3+a) * b$. Однако, при такой записи мы не знаем, что Паскаль будет делать раньше - делить $(1 + a / (2+a*b))$ на $(3+a)$ или умножать $(3+a)$ на b. А от этого зависит результат. Добавим для верности пару скобок: $((1 + a / (2+a*b)) / (3+a)) * b$. Теперь все в порядке.

К сожалению, в выражениях разрешается писать только круглые скобки. Квадратные и фигурные запрещены. От этого сложные выражения с большим количеством скобок на глаз воспринимаются с трудом, так как трудно для конкретной скобки увидеть ее законную пару. В этом случае я могу посоветовать идти "от малого к большому", то есть сначала заметить самые малые из взятых в скобки фрагменты выражения (у нас это $3+a$ и $2+a*b$). После этого будет уже легче заметить более крупные фрагменты, такие как $1 + a / (2+a*b)$, и т.д.

Запись десятичных дробей. Почти во всех языках программирования и уж, конечно, в Паскале, в десятичных дробях принято вместо запятой ставить точку. Пример: 62.8 - шестьдесят две целых восемь десятых.

Математические функции. Кроме четырех действий арифметики Паскаль может выполнять и другие математические действия, например, возведение в квадрат, для чего имеется специальная функция - *Sqr*. На уроке математики мы обозначаем показатели степени маленькими цифрами и буквами. На компьютере такие цифры и буквы вводить не всегда возможно, поэтому в Паскале принято другое обозначение. Например, пять в квадрате обозначается так - *Sqr(5)*, a+b в квадрате так - *Sqr(a+b)*. Здесь *Sqr* - сокращение от английского слова square - квадрат. То, что нужно возвести в квадрат, записывается в скобках.

Приведу неполный список математических функций Паскаля:

ДЕЙСТВИЕ	РЕЗУЛЬТАТ	СМЫСЛ
Sqr (5)	25	возведение в квадрат
Sqrt (25)	5	корень квадратный
Pi	3.1415...	число пи
Frac (23.192)	0.192	дробная часть числа
Int (3.98)	3.0	целая часть числа
Round (5.8)	6	округление
Abs (-20)	20	абсолютная величина (модуль) числа
Random	0.73088	случайное число из диапазона (0 - 1)
Random (200)	106	случайное целое число из диапазона (0 - 199)

Кроме этого, имеются функции *sin, cos, arctan, exp, ln* и процедура *Randomize*. К сожалению, в Паскале нет специальной функции для возведения в произвольную степень.

Примеры:

Выражение	**Sqr(2+1)**	при вычислении даст **9**
Выражение	**10+Sqr(2+1)**	при вычислении даст **19**
Выражение	**1+Abs(5-8)**	при вычислении даст **4**
Выражение	**Sqr(2)+Sqrt(35+1)**	при вычислении даст **10**
Выражение	**Sqrt(8+Int(41.5))**	при вычислении даст **7**
Выражение	**21 div (Round(Pi+1))**	при вычислении даст **5**

Задание 11: Определите без компьютера, что напечатает данная программа:

```
VAR a,b: Integer;
BEGIN
   a:=(Sqr(2)+1)*(20- Sqr(2*2))-11;
   b:=11 div (a-4);
   WriteLn (Sqr(a)+b-1)
END.
```

4.10. Вещественные числа в Паскале

Вот ***ошибочная*** программа:

```
VAR a,b,y : Integer;
BEGIN
   a:=10;   b:=6;
   y:= a / b;
   WriteLn (y)
END.
```

Паскаль откажется выполнять эту программу, так как знает, что при делении целого на целое результат может получиться дробный, а это значит, что в ячейку y придется записывать дробное число. Но описание *VAR a,b,y : Integer* запрещает это делать. Вот вам еще один пример придирчивости Паскаля. Если бы мы даже вместо *b:=6* написали *b:=2* , все равно Паскаль отказался бы делить.

Что же делать? Конечно же, Паскаль предлагает простой выход. Программист имеет право любую переменную описать не как целую (*Integer*), а как вещественную (*Real*). В этом случае переменная имеет право принимать любые целые и дробные значения. Вот ***правильная*** программа:

```
VAR  a,b     :Integer;
     y       :Real;
BEGIN
   a:=10;   b:=6;
   y:=a / b;
   WriteLn (y)
END.
```

По-английски *Real* читается " 'риэл", переводится "вещественный". Под переменную типа Real Паскаль отводит в памяти ячейку размером в 6 байтов.

Что мы увидим на экране в результате выполния оператора *WriteLn (y)*? Если вы думаете, что 1.666666 , то ошибаетесь. Переменные, описанные как Real, Паскаль выводит на экран в так называемом экспоненциальном формате (виде), с первого взгляда непонятном. Более подробно об этом и других форматах мы поговорим в 14.5, а пока, чтобы заставить Паскаль выводить вещественные числа в обычном, понятном виде, допишем кое-что в оператор вывода - *WriteLn (y :8:3)*. Это значит, что мы хотим численное значение переменной *y* типа Real видеть на экране в привычном виде с 3 знаками после десятичной точки, а всё изображение числа не должно занимать больше 8 символов, включая целую часть, дробную часть, знак и десятичную точку. Этот оператор напечатает на экране **1.667**. Здесь напечатано действительно 8 символов (три пробела, предшествующие единице, видны вам, как пустое место). Вместо 8 и 3 в операторе программист может писать любые имеющие смысл числа.

Поэкспериментируйте: *(y :38:3), (y :20:10), ('Результат равен', y :8:3).*

Три совета

Дорогой читатель! Если у вас под рукой есть компьютер, то вот вам три моих совета, по своей силе приближающихся к непререкаемым приказам:

1. Программы, которые вы видите в книге, вводите в компьютер и выполняйте их, даже если они кажутся вам почти понятными. В ряде случаев вы получите неожиданные результаты, из чего сделаете вывод, что программы эти вы поняли не до конца.

2. В каждой из этих программ экспериментируйте, то есть разными способами изменяйте в них то, что я как раз в этот момент объясняю. Например, если я объясняю оператор **for i:=1 to 5,** ***пробуйте*** **for i:=1 to 10.**

3. Выполняйте и сверяйте с ответом все задания. Это, конечно, главный совет из трех. Учтите, что сверенная с ответом правильно работающая программа – ваша победа, сверенная с ответом неправильно работающая программа – временное поражение, отказ от сверки - разгром.

Если вы пожалеете времени и пренебрежете этими советами, то через несколько страниц можете обнаружить трудности в понимании материала и вскоре не сможете правильно составить большинство программ.

4.11. Порядок составления простой программы

Задача:

Известны размеры спичечной коробки: высота - 12.41 см., ширина - 8 см., толщина - 5 см. Вычислить площадь основания коробки и ее объем .

Порядок составления программы:

1. Программист сам должен знать решение задачи. Ведь программа - это инструкция по ее решению. Нельзя давать инструкцию, не зная, как решать.

В нашем случае программист должен знать формулы для вычисления площади основания коробки и ее объема : площадь = ширина x толщина , объем = площадь x высота .

2. Нужно придумать имена переменным. Имя переменной должно говорить о ее смысле. Если смыслом является ширина коробки, то не ленитесь и не называйте ее a, потому что через полгода, разбираясь в своей полузабытой программе, вы будете долго тереть лоб и думать – Что, черт возьми, я обозначил через a? Называйте ее *shirina* (если вы не знаете английского) или, к примеру, *width* (если знаете). Так делают все профессиональные программисты (а они, как известно, терпеть не могут трудиться зря, значит, зачем-то это им нужно).

Удовлетворимся такими именами:

shirina	-	ширина
tol	-	толщина
visota	-	высота
pl	-	площадь
V	-	объем

3. Нужно определить, какого типа будут переменные. Поскольку ширина и толщина - целые, то и площадь будет целой. Высота и, следовательно, объем - вещественные. Первые две строки программы будут такими:

```
VAR     shirina,tol,pl : Integer;
        visota,V       : Real;
```

4. Перед вычислениями нужно задать исходные данные решения задачи. Вот следующие две строки программы:

```
BEGIN
        shirina:=8;   tol:=5;   visota:=12.41;
```

5. Теперь нужно задать компьютеру действия, которые нужно проделать с исходными данными, чтобы получить результат.

```
        pl := shirina * tol;
        V  := pl * visota;
```

6. После получения результата его нужно напечатать. Действительно, все операторы присваивания компьютер выполняет "в уме". После их выполнения в ячейках памяти pl и V будут находиться числовые результаты решения задачи. Чтобы их узнать, человек в нашем случае может использовать оператор WriteLn, после чего программу можно заканчивать:

```
        WriteLn (pl,'    ', V  :10:3)
END.
```

Обратите внимание, что поскольку переменная V имеет тип Real, для ее вывода мы использовали формат (см.4.10).

Вот как будет выглядеть наша программа целиком:

```
VAR  shirina,tol,pl     :Integer;
     visota,V           :Real;
BEGIN
     shirina:=8;   tol:=5;   visota:=12.41;
     pl := shirina * tol;
     V  := pl * visota;
     WriteLn (pl,'    ', V  :10:3)
END.
```

Программа напечатает два числа: 40 и 496.400 .

Эту задачу можно было бы решить и гораздо более короткой программой:

```
BEGIN    WriteLn (8 * 5, '     ' , 8 * 5 * 12.41 :10:3)    END.
```

А еще быстрее эту задачу решить в уме. Однако, соблюдение приведенного мной порядка составления программы облегчит вам в дальнейшем программирование реальных задач для компьютера.

Задания 12-14:

Написать программы для решения следующих задач:

12) В углу прямоугольного двора размером 50x30 стоит прямоугольный дом размером 20x10. Подсчитать площадь дома, свободную площадь двора и длину забора. Примечание: в углу, где дом, забора нет.

13) Радиус окружности равен 800. Вычислить длину окружности и площадь круга. Результаты печатать с 5 знаками после десятичной точки.

14) Автомобиль 3 часа ехал со скоростью 80 км/час и 2 часа со скоростью 90 км/час. Вычислить среднюю скорость автомобиля (она равна суммарному пути, деленному на суммарное время).

4.12. Операторы ввода данных ReadLn и Read.

Задача: Сложить два числа - 20 и 16.
Сравним две программы решения этой задачи:

```
VAR a,b : Integer;
BEGIN
  a:=20;   b:=16;
  WriteLn (a+b)
END.
```

```
VAR a,b : Integer;
BEGIN
  ReadLn (a,b);
  WriteLn (a+b)
END.
```

Программы отличаются только одной строкой. Первая программа не требует пояснений - она печатает число 36. Во второй программе нигде не сказано, чему равны a и b, а вместо этого включен оператор ReadLn. Поговорим о нем.

ReadLn читается "рид'лайн", переводится "читай строку". Он приказывает компьютеру остановиться и ждать, когда человек введет с клавиатуры определенную информацию, после чего продолжить работу. В частности, *ReadLn (a,b)* будет ждать ввода двух целых чисел.

Таким образом, если первая программа после запуска будет работать без остановки до самого конца и без хлопот выдаст результат, то вторая программа на операторе ReadLn остановится и будет ждать. Во время этого ожидания человек должен на клавиатуре набрать число 20 (так как первым в списке оператора ReadLn стоит a), затем нажать клавишу пробела, затем набрать 16 и нажать клавишу Enter. Паскаль воспринимает нажатие пробела, как сигнал человека о том, что закончен набор на клавиатуре одного числа и сейчас начнется набор другого. После набора на клавиатуре последнего числа необходимо нажать клавишу Enter в знак того, что ввод чисел для данного оператора ReadLn закончен и компьютер может продолжать работу. В соответствии с этим компьютер сразу же после нажатия Enter прекращает ожидание и прежде всего направляет число 20 в память, в ячейку a, число же 16 - в ячейку b. На этом он считает выполнение оператора ReadLn законченным и переходит к следующему оператору - WriteLn. В результате будет напечатано число 36.

Таким образом, обе программы делают одно и то же. Зачем же тогда применять ReadLn вместо оператора присваивания? Ведь первая программа понятней, да и работает без остановки. Одна из причин в том, что программа с ReadLn гораздо универсальнее, "свободнее": если первая программа решает задачу сложения только двух конкретных чисел, то вторая программа складывает два любых числа. Вторая причина в том, что программа с ReadLn позволяет программисту во время написания программы не задумываться над конкретными значениями исходных данных, оставляя эту головную боль на момент выполнения программы. Но самая главная причина в том, что ReadLn позволяет человеку общаться с компьютером, вести с ним диалог во время выполнения программы.

В подтверждение важности первой причины напишем программу для решения следующей **задачи:** В зоопарке три слона и довольно много кроликов, причем количество кроликов часто меняется. Слону положено съедать в сутки сто морковок, а кролику - две. Каждое утро служитель зоопарка сообщает компьютеру количество кроликов. Компьютер в ответ на это должен сообщить служителю общее количество морковок, которые сегодня нужно скормить кроликам и слонам.

Придумаем имена переменным величинам:

kol_krol	-	количество кроликов в зоопарке
kol_slon	-	количество слонов в зоопарке
norma_krol	-	сколько морковок в день положено кролику
norma_slon	-	сколько морковок в день положено слону
vsego	-	сколько всего требуется морковок

А теперь напишем программу:

```
VAR kol_krol,kol_slon,norma_krol,norma_slon,vsego   :Integer;
BEGIN
    norma_krol:=2;
    norma_slon:=100;
    ReadLn (kol_krol);
    kol_slon:=3;
    vsego := norma_krol * kol_krol  +  norma_slon * kol_slon;
    WriteLn (vsego)
END.
```

Написав программу, программист вводит ее в компьютер, отлаживает и записывает на диск. На этом его миссия закончена. Утром служитель, пересчитав кроликов и найдя, что их 60 штук, подходит к компьютеру и запускает программу на выполнение.

Компьютер, выполнив автоматически первые два оператора (*norma_krol:=2* и *norma_slon:=100*), останавливается на операторе ReadLn. Служитель вводит число 60, после чего компьютер посылает это число в ячейку *kol_krol* и переходит к выполнению следующего оператора (*kol_slon:=3*). В конце концов на мониторе появится ответ: 420.

Вот схематическое изображение процесса выполнения программы:

ПОРЯДОК ИСПОЛНЕНИЯ ОПЕРАТОРОВ	ЧТО НАХОДИТСЯ В ЯЧЕЙКАХ ПАМЯТИ				
	kol_krol	kol_slon	norma_krol	norma_slon	vsego
norma_krol:=2	?	?	2	?	?
norma_slon:=100	?	?	2	100	?
ReadLn (kol_krol)	60	?	2	100	?
kol_slon:=3	60	3	2	100	?
vsego:=norma_krol	60	3	2	100	420
WriteLn (vsego)	60	3	2	100	420

На следующее утро, обнаружив, что 5 кроликов продано другому зоопарку, служитель запускает ту же самую программу, вводит число 55 и получает ответ - 410.

На этом несколько фантастичном примере я хотел показать, что применение ReadLn позволяет создавать программы, которые, оставаясь приятно неизменными, позволяют легко решать задачу в любое время для любых значений исходных данных. Можно было бы пойти по другому пути - вместо ReadLn использовать оператор присваивания, например *kol_krol:=60*. Но в этом случае программист каждое утро должен был бы бежать в зоопарк, чтобы исправлять в программе этот оператор присваивания.

Оператор ReadLn можно писать и без скобок, просто так: ReadLn . Выполняя оператор в такой записи, компьютер остановится и будет ждать, но не ввода какой-то информации, а просто нажатия на клавишу Enter. Таким образом, это просто оператор создания паузы в процессе выполнения программы. О том, зачем нужны паузы, поговорим чуть ниже.

Кроме оператора ReadLn для ввода данных применяется также оператор Read. Для начинающего программиста различия в их применении несущественны. Мы будем пока употреблять только ReadLn. Оператор Read без скобок паузу не создает.

4.13. Интерфейс пользователя

Когда служитель запускает программу и она делает паузу на операторе *ReadLn(kol_krol)*, служитель видит перед собой пустой экран монитора, на котором нет никаких намеков на приглашение вводить какую-либо информацию. Посторонний человек ни за что и не догадается, что компьютер чего-то ждет. Это не очень удобно. Было бы гораздо лучше, если бы на экране все-таки можно было в нужный момент видеть подходящее приглашение, чтобы служитель с раннего утра чего-нибудь не перепутал.

Это же касается и выдачи результатов. На пустом экране появляется сухое число 420. Посторонний человек ни за что не поймет, какой оно имеет смысл: то ли это 420 рублей, то ли 420 зоопарков. Говорят, что у нашей программы неудобный **интерфейс пользователя**, то есть человеку, использующему нашу программу, неудобно с ней общаться. Слово "интерфейс" можно перевести, как "взаимодействие", в данном случае взаимодействие человека с компьютером.

Дополним чуть-чуть нашу программу, чтобы интерфейс стал более удобным:

```
VAR  kol_krol,kol_slon,norma_krol,norma_slon,vsego : Integer;
BEGIN
    norma_krol:=2;
    norma_slon:=100;
    WriteLn ( 'Введите, пожалуйста, количество кроликов' );
    ReadLn ( kol_krol);
    kol_slon:=3;
    vsego := norma_krol *  kol_krol  +  norma_slon *  kol_slon;
    WriteLn( 'Вам всего понадобится  ' ,vsego, '  морковок' );
    ReadLn
END.
```

Эта программа будет работать точно так же, как и предыдущая, с тем отличием, что во время паузы, вызванной оператором *ReadLn (kol_krol)*, на экране будет гореть удобная надпись -

Введите, пожалуйста, количество кроликов

а результат будет выведен на экран в виде -

Вам всего понадобится 420 морковок

Оператор ReadLn без скобок в конце программы нужен для нейтрализации одной неприятной особенности в работе Паскаля. Дело в том, что выполнив программу, Паскаль торопится погасить экран с результатами решения задачи и делает это так быстро, что человек просто не успевает эти результаты разглядеть. Оператор ReadLn, поставленный после оператора WriteLn, выводящего результаты на экран, задает паузу. Во время этой паузы экран не гаснет, так как программа еще не выполнилась до конца, и человек может спокойно разглядеть результаты. После этого он нажатием клавиши Enter позволит компьютеру продолжить выполнение программы (в нашем случае после ReadLn стоит END с точкой, поэтому программа завершится).

Часто, впрочем, можно обойтись и без ReadLn. Нажав пару клавиш на клавиатуре (*Alt-F5*), мы можем снова зажечь погасший экран с результатами. В дальнейших примерах я буду для экономии места и "смысла" обходиться без ReadLn.

Задания 15-16:

Написать с использованием интерфейса программы решения задач:

15) Длина одной стороны треугольника равна 20. Длины двух других сторон будут известны только после запуска программы на выполнение. Вычислить периметр треугольника.

16) В компьютер вводятся путь, пройденный телом, и скорость тела. Найти время движения тела.

4.14. Строковые переменные

Сравним две программы:

```
VAR a : Integer;
BEGIN
  a:=98;
  WriteLn (a)
END.
```

```
VAR a : String;
BEGIN
  a:='Привет всем!';
  WriteLn (a)
END.
```

В первой программе описание *VAR a : Integer* говорит о том, что переменная *a* обязана иметь числовое значение, а оператор *a:=98* записывает в ячейку *a* число 98.

Во второй программе описание *VAR a : String* говорит о том, что переменная a обязана иметь строковое (текстовое) значение, то есть ее значением будет не число, а произвольная цепочка символов, например, *Привет всем!* или *pnH2H(*fD6:u* . Оператор *a:='Привет всем!'* записывает в ячейку *a* строку *Привет всем!* . Оператор *WriteLn (a)*, поскольку он обязан всегда выводить на экран содержимое ячейки *a*, выведет на экран текст *Привет всем!*

Обратите внимание, что в программе текст должен браться в кавычки, а в памяти он хранится без кавычек и на экран выводится без кавычек.

Слово String читается "стринг", переводится "строка".

Какой смысл переменным иметь текстовое значение, выяснится в следующем параграфе.

Информация в ячейке памяти под строковую переменную может в процессе выполнения программы меняться точно так же, как и в ячейке для числовой переменной. Например, при выполнении фрагмента

```
a:='Минуточку!'; WriteLn(a); a:='Здравствуйте!'; a:='До свидания!'; WriteLn (a)
```

в ячейке *a* будут по очереди появляться строки

Минуточку! Здравствуйте! До свидания!

а на экран будут выведены строки

Минуточку! До свидания!

Строковую переменную можно задавать не только оператором присваивания, но и оператором ReadLn. Пример:

```
VAR a : String;
BEGIN
   WriteLn ('Введите какое-нибудь слово');
   ReadLn (a);
   WriteLn ('Вы ввели слово  ' ,a)
END.
```

Во время паузы, вызванной оператором ReadLn, вы должны ввести какой-нибудь набор символов, например *Изнакурнож* , и затем нажать клавишу Enter в знак того, что ввод закончен. В результате на экране будет напечатан текст:

Вы ввели слово Изнакурнож

4.15. Диалог с компьютером

Напишем программу, которая осуществляла бы такой диалог человека с компьютером:

КОМПЬЮТЕР ВЫВОДИТ НА ЭКРАН:	Здравствуй, я компьютер, а тебя как зовут?
ЧЕЛОВЕК ВВОДИТ С КЛАВИАТУРЫ:	*Коля*
КОМПЬЮТЕР ВЫВОДИТ НА ЭКРАН:	Очень приятно, Коля. .Сколько тебе лет?
ЧЕЛОВЕК ВВОДИТ С КЛАВИАТУРЫ:	*16*
КОМПЬЮТЕР ВЫВОДИТ НА ЭКРАН:	Ого!Целых 16 лет!Ты уже совсем взрослый!

Вот программа:

```
VAR  imya      :String;
     vozrast   :Integer;
BEGIN
   WriteLn ('Здравствуй, я компьютер, а тебя как зовут?');
   ReadLn (imya);
   WriteLn ('Очень приятно, ', imya, ' . Сколько тебе лет?');
   ReadLn (vozrast);
   WriteLn ('Ого! Целых ' ,vozrast, ' лет! Ты уже совсем взрослый!')
END.
```

Диалог будет отличаться только той информацией, которую вводит человек. Так, в другой раз по этой же программе будет осуществлен следующий диалог:

КОМПЬЮТЕР:	Здравствуй, я компьютер, а тебя как зовут?
ЧЕЛОВЕК :	*Фантомас!*
КОМПЬЮТЕР:	Очень приятно, Фантомас! . Сколько тебе лет?
ЧЕЛОВЕК:	*100*
КОМПЬЮТЕР:	Ого! Целых 100 лет! Ты уже совсем взрослый!

Не подумайте, что эта программа очень умная. Она совершенно не анализирует, какую информацию человек ввел с клавиатуры. Поэтому с ней возможен и такой диалог:

КОМПЬЮТЕР:	Здравствуй, я компьютер, а тебя как зовут?
ЧЕЛОВЕК:	*Сгинь с моих глаз!*
КОМПЬЮТЕР:	Очень приятно, Сгинь с моих глаз! . Сколько тебе лет?
ЧЕЛОВЕК:	*-2*
КОМПЬЮТЕР:	Ого! Целых -2 лет! Ты уже совсем взрослый!

Задание 17: Напишите программу для следующей задачи: Компьютер спрашивает названия двух планет, радиусы их орбит (в миллионах километров) и скорости движения по орбите (в миллионах километров в сутки). Компьютер вычисляет продолжительность года на планетах и выдает результат в таком виде: *Продолжительность года на планете Земля – 365 суток, а на планете Эоэлла – 12 суток.*

4.16. Константы

Кроме переменных величин в тексте программы мы встречаем **константы**. Это те значения, которые или присванваются переменным, или встречаются в выражениях, или сравниваются с выражениями. Например:

x:=25	здесь числовая константа -	25
slovo:=’Волга’	здесь строковая константа -	Волга
simvol:=’!’	здесь символьная константа -	!
y:=(x+1) / (200*x - 0.3)	здесь числовые константы -	1 и 200 и 0.3

Глава 5. Разветвляющиеся программы

5.1. Условный оператор IF или как компьютер делает выбор

Идею разветвления в программе я изложил в 2.8. Сейчас добавлю только, что вся мыслительная деятельность во всех программах (в том числе и той, что выиграла в шахматы у Каспарова) осуществляется только при помощи ветвления (выбора).

Теперь посмотрим, как писать разветвляющиеся программы на Паскале.

Выучим сначала три английских слова:

if	читается "иф"	переводится "если"
then	читается "зэн"	переводится "то"
else	читается "элз"	переводится "иначе"

Теперь приведем пример нового для вас оператора:

IF a=28 THEN WriteLn (f) ELSE k:=44

Переводится он так:

ЕСЛИ a=28 ТО печатай f ИНАЧЕ присвой переменной k значение 44.

Другими словами, мы предлагаем компьютеру сначала подумать, правда ли, что a=28 , и если правда, то выполнить оператор *WriteLn (f)*, в противном случае выполнить оператор *k:=44* . Таким образом, мы с вами впервые написали оператор, при выполнении которого компьютер не просто выполняет, что приказано, а сначала думает и делает выбор (пока одного из двух).

Мы видим, что оператор *if* включает в себя другие операторы, которые выполняются или не выполняются в зависимости от какого-то условия. Чтобы понять, зачем может пригодиться оператор *if*, рассмотрим следующую задачу.

Задача 1. Компьютер должен перемножить два числа - 167 и 121. Если их произведение превышает 2000, то компьютер должен напечатать текст ПРОИЗВЕДЕНИЕ БОЛЬШОЕ, иначе текст ПРОИЗВЕДЕНИЕ МАЛЕНЬКОЕ. После этого компьютер в любом случае должен напечатать само произведение.

Программа:

```
VAR a,b,y :Integer;
BEGIN
   a:=167;
   b:=121;
   y:=a*b;
   if  y>20000     then WriteLn ('ПРОИЗВЕДЕНИЕ  БОЛЬШОЕ')
                   else  WriteLn ('ПРОИЗВЕДЕНИЕ МАЛЕНЬКОЕ');
   WriteLn (y)
END.
```

Пояснение: В программе 5 операторов, последний – *WriteLn (y)*. Поскольку эти 5 операторов выполняются по порядку, то он выполнится обязательно.

Задача 2. В компьютер вводятся два произвольных положительных числа - длины сторон двух кубиков. Компьютер должен подсчитать объем одного кубика - большего по размеру .

Обозначим *a1* - сторону одного кубика, *a2* - сторону другого, *bol* - сторону большего кубика, *V* - объем кубика. Приведем три варианта программы:

ВАРИАНТ 1	**VAR** a1,a2 : Real; **BEGIN** ReadLn (a1,a2); **if** a1>a2 **then** WriteLn (a1*a1*a1 : 15:5) **else** WriteLn (a2*a2*a2 : 15:5) **END.**
ВАРИАНТ 2	**VAR** a1,a2,V : Real; **BEGIN** ReadLn (a1,a2); **if** a1>a2 **then** V:=a1*a1*a1 **else** V:=a2*a2*a2; WriteLn (V : 15:5) **END**.
ВАРИАНТ 3	**VAR** a1,a2,bol,V : Real; **BEGIN** ReadLn (a1,a2); **if** a1>a2 **then** bol:=a1 **else** bol:=a2; V:=bol*bol*bol; WriteLn (V : 15:5) **END**.

Поясним последний вариант. Программа состоит из четырех операторов, которые выполняются в порядке записи. Первым после запуска выполняется оператор *ReadLn (a1,a2)*, который ждет от нас ввода двух чисел. Пусть мы сегодня ввели числа 3 и 2. Компьютер понимает, что *a1* равно 3, *a2* равно 2, и переходит к выполнению оператора *if*. Он видит, что 3>2, и поэтому выполняет оператор *bol:=a1*, а оператор *bol:=a2* не выполняет. В результате в ячейке *bol* оказывается число 3. Затем компьютер переходит к следующему оператору - *V:=bol*bol*bol* и вычисляет V=3*3*3=27. Следующий оператор *WriteLn (V : 15:5)* печатает число 27.00000 .

Если завтра мы запустим эту же программу и введем числа 6 и 10, то компьютер увидит, что утверждение 6>10 ложно, и поэтому выполнит оператор *bol:=a2*, а оператор *bol:=a1* выполнять не станет. В результате в ячейке *bol* окажется число 10 и будет напечатано число 1000**.**00000 .

> *А теперь, дорогой читатель, вам пришла пора освоить пошаговый режим выполнения программы на компьютере. В обычном режиме компьютер выполняет программу настолько быстро, что вы не успеваете заметить, в каком порядке он выполняет отдельные операторы программы, а без этого часто невозможно понять ее логику. Пошаговый режим заставляет компьютер при выполнении программы задерживаться на каждой строке программы столько, сколько нужно человеку. Сейчас вам необходимо проделать то, что сказано в части IV в пункте «Пошаговый режим» параграфа «Исправление ошибок. Отладка программы».*

Итак, если паровая машина избавила человека от тяжелого физического труда, то оператор *if* избавил человека от тяжелого умственного труда, в нашем случае - от необходимости решать, какое из двух чисел больше другого.

Оператор *if* можно записывать и без части *else*. Например, *if s<t then w:=a+1*. Это означает, что если *s<t*, то нужно выполнить оператор *w:=a+1*, в противном случае ничего не делать, а просто перейти к следующему оператору.

Для примера рассмотрим простейшую задачу: В компьютер вводится слово. Компьютер должен просто распечатать его. Однако, если введенным словом будет "колхозник", то компьютер должен напечатать вместо него слово "фермер".

Вот как будет выглядеть наша программа-"цензор":

```
VAR  Slovo : String;
BEGIN
   ReadLn (Slovo);    { переменная Slovo будет иметь  значением строку символов, введенных с клавиатуры }
   if  Slovo = 'колхозник'  then  Slovo := 'фермер';
   WriteLn (Slovo)
END.
```

5.2. Правила записи оператора IF

У каждого человеческого языка есть своя грамматика, включающая в себя правила, по которым должны выстраиваться в цепочку элементы языка, чтобы получилось правильное предложение. Совокупность этих правил образует часть грамматики, называемую синтаксисом. В языках программирования тоже есть предложения. Такими предложениями здесь являются операторы. Следовательно, у языков программирования тоже должен быть свой синтаксис, который описывает правила, по которым записываются операторы языка и из операторов составляется программа. После того, как человек запускает программу на выполнение, любая порядочная среда программирования прежде, чем действительно выполнять ее, сначала проверит, нет ли в ней синтаксических ошибок, и если есть, то программу выполнять не будет, а выдаст сообщение, указывающее человеку, в чем ошибка.

Пока мы не готовы воспринять полный синтаксис оператора if Паскаля (вы найдете его в 14.8), однако уже сейчас нам хотелось бы писать без ошибок. Для этого, скрепя сердце, будем использовать сокращенный синтаксис. Поясним его в виде синтаксической схемы:

IF условие **THEN** оператор **ELSE** оператор

Как понимать эту схему? Ее следует понимать, как образец, шаблон записи оператора, указывающий порядок, в котором оператор записывается из отдельных слов. Слова, которые в схеме я записал жирными заглавными буквами, при записи оператора просто копируются. Вместо слов, которые в схеме записаны строчными буквами, нужно при записи оператора подставить то, что они означают. Поясним, что обозначают эти слова.

оператор	любой оператор Паскаля
условие	пока под условием будем понимать два арифметических выражения, соединенных знаком сравнения, или условие равенства или неравенства строк, как это показано на примере в 5.1
выражение	что такое выражение, было пояснено в 4.9
знак сравнения	знаков сравнения шесть: > больше >= больше или равно = равно < меньше <= меньше или равно <> не равно

Пример: **if** 5*a+4 <= a*b **then** WriteLn (b) **else** a:=b+5 Здесь

WriteLn (b) - один оператор,
a:=b+5 - другой оператор,
5*a+4 <= a*b - условие,
5*a+4 - одно выражение,
a*b - другое выражение,
<= - знак сравнения.

Вспомним правило расстановки точек с запятыми. Они применяются для того, чтобы отделять друг от друга операторы, выполняющиеся друг за другом. Поэтому и после оператора *if* мы тоже ставили точку с запятой, если после него шел какой-нибудь оператор. Распространенная привычка начинающих программистов - автоматически ставить точку с запятой после любого оператора, независимо от того, что после него стоит - другой оператор или же служебное слово, например else или end. Так вот, перед *end* точку с запятой ставить не возбраняется, а

перед ELSE точку с запятой ставить запрещено.

В п.5.1 вы уже видели, что оператор *if* можно записывать в краткой форме:

IF условие **THEN** оператор

Таким образом, это уже вторая синтаксическая схема, касающаяся одного оператора. Удобно же весь синтаксис оператора иметь перед глазами в одной схеме. Соединим две схемы в одну. Вот эта схема:

IF условие **THEN** оператор **[ELSE** оператор **]**

Квадратные скобки здесь означают, что их содержимое можно писать, а можно и не писать в операторе.

Полезное замечание: Вычисляя выражения, стоящие в условии оператора if, Паскаль не записывает их значения в память. Например, после выполнения фрагмента - *b:=6; if b+1>0 then s:=20* **- в ячейке** *b* **будет храниться** *6*, **а не** *7*. **То же относится и к выражениям из оператора WriteLn. Например:** *b:=6; WriteLn (b+1)* **. И здесь тоже в ячейке** *b* **останется храниться** *6*, **а не** *7*. **И вообще, информация в ячейках памяти не меняется при вычислении выражений.**

Примеры работы оператора *if*:

ФРАГМЕНТ ПРОГРАММЫ	ЧТО НА ЭКРАНЕ
a:=10; **if** a>2 **then** WriteLn ('!!!') **else** WriteLn ('!')	!!!
a:=4; **if** a>5 **then** a:=a+10 **else** a:=a-1; WriteLn (a)	3
s:=6; **if** s-8<0 **then** s:=s+10; WriteLn (s)	16
s:=6; **if** s<0 **then** s:=s+10; s:=s+1; WriteLn (s)	7

Пояснение: Обратите внимание, что в последнем примере оператор *if* кончается оператором *s:=s+10*, а не *s:=s+1*. Поэтому оператор *s:=s+1* будет выполняться всегда, независимо от величины *s*.

Задания 18-20:

Определить без компьютера, что будет напечатано при выполнении следующих фрагментов программ:

18. k:=20; k:=k+10; **if** k+10<>30 **then** k:=8 **else** k:=k-1; WriteLn (k)

19. k:=20; k:=k+10; **if** k+10 = 30 **then** k:=8 **else** k:=k-1; WriteLn (k)

20. p:=1; **if** p>0 **then** p:=p+5; Write (p); **if** p>10 **then** p:=p+1; Write (p)

Задания 21-23:

21. В компьютер вводятся два числа. Если первое больше второго, то вычислить их сумму, иначе - произведение. После этого компьютер должен напечатать текст ЗАДАЧА РЕШЕНА.

22. В компьютер вводятся длины трех отрезков. Компьютер должен ответить на вопрос, правда ли, что первый отрезок не слишком длинен, чтобы образовать с другими двумя отрезками треугольник.

Указание: Для этого его длина должна быть меньше суммы длин двух других отрезков. Замечание: Пока не думайте о том, что слишком длинными могут быть второй или третий отрезки. Об этом – задание из 5.5.

23. Дракон каждый год отращивает по три головы, но после того, как ему исполнится 100 лет - только по две. Сколько голов и глаз у дракона, которому N лет?

5.3. Составной оператор

Вот фрагмент программы, которая складывает два числа:

```
WriteLn ('Введите два числа');
ReadLn (a,b);
WriteLn ('Сумма равна ' ,a+b)
```

Вот фрагмент программы, которая возводит число в квадрат:

```
WriteLn ('Введите число');
ReadLn (a);
WriteLn ('Квадрат числа равен ' ,a*a)
```

Пусть мы хотим сделать программу, которая бы по желанию пользователя или складывала два числа, или возводила одно число в квадрат, то есть выполняла или первый или второй фрагмент.

Начинаться наша программа могла бы примерно так:

```
VAR  Otvet : String;  . . . . .
BEGIN
   WriteLn ('Чем займемся – сложением или возведением в квадрат?');
   ReadLn (Otvet);
   if  Otvet = 'сложением'  then  . . . .  else . . . .
```

Здесь после then мы должны бы вставить первый фрагмент, а после else второй. Однако, тут возникает проблема. Каждый из фрагментов состоит из нескольких операторов, а синтаксис оператора if разрешает ставить после then и else только по одному оператору. Чтобы преодолеть эту трудность, в Паскале есть средство превратить последовательность записанных друг за другом операторов формально в один оператор. Для этого последовательность заключается между словами begin и end и получившаяся конструкция называется **составным оператором**.

Вот первый фрагмент в виде составного оператора:

```
begin
   WriteLn ('Введите два числа');
   ReadLn (a,b);
   WriteLn ('Сумма равна ' ,a+b)
end
```

Компьютер выполняет составной оператор неотличимо от последовательности операторов, из которых он состоит, однако титул составного оператора позволяет ему считаться одним оператором и поэтому быть вхожим в те места программы, в которые разрешен вход только единичным операторам, а не их последовательностям, как, например, имеет место в нашем случае с оператором *if*.

Вот какая получится программа для нашей задачи:

```
VAR  Otvet  :String;
     a,b    :Integer;
BEGIN
   WriteLn ('Чем займемся - сложением или возведением в квадрат?');
   ReadLn (Otvet);
   if  Otvet = 'сложением'
      then
         begin WriteLn ('Введите два числа');
               ReadLn (a,b);
               WriteLn ('Сумма равна ' ,a+b)
         end
      else
         begin WriteLn ('Введите число');
               ReadLn (a);
               WriteLn ('Квадрат числа равен ' ,a*a)
         end;
   WriteLn ('Счет завершен')
END.
```

У нашей программы есть недостаток. Если при ответе на вопрос компьютера мы чуть-чуть ошибемся, например, ответим не "сложением", а "сложение", компьютер будет выполнять возведение в квадрат, так как в условии оператора if сравниваемые строки должны совпадать полностью. Научившись выполнять операции над строками, вы научитесь избегать таких ситуаций.

Задание 24: Видоизменить диалог с компьютером, начатый в 4.15. Пусть компьютер, узнав возраст человека, дальнейшую беседу ведет по двум вариантам. Если возраст больше 17, то компьютер должен задать вопрос: "В каком институте ты учишься?" и получив ответ, глубокомысленно заметить "Хороший институт". Если же возраст меньше или равен 17, то соответственно - "В какой школе ты учишься?" и "Неплохая школа". После этого, каков бы ни был вариант, компьютер должен попрощаться: "До следующей встречи!".

5.4. Ступенчатая запись программы

Обратите внимание на то, на что не обращает внимания компьютер, а именно на отступы от левого края листа в записи строк программы из 5.3. Строки VAR, BEGIN и END записаны без отступа. Между словами BEGIN и END записаны четыре оператора: WriteLn, ReadLn, if и WriteLn. Все они выполняются по порядку, один за другим, поэтому каждый из них записан с одинаковым отступом. Если оператор сложный, то есть включает в себя другие операторы (мы знаем пока два таких оператора - if и составной), то составляющие его операторы записываются еще правее. Слова, составляющие пару (then и else, begin и end) записываются друг под другом.

Сделано все это для удобства чтения программы, для того, чтобы глаз мог сразу же уловить структуру программы, а именно, из каких частей состоит как сама программа, так и каждый из элементов, ее составляющих. Впрочем, вам с первого взгляда может показаться, что такая запись, наоборот, неудобна для чтения. Однако, заметьте, что ступенчатая запись принята во всем мире и глаза профессиональных программистов привыкли именно к ней. Настолько привыкли, что программа, записанная без соблюдения ступенчатого стиля, вызывает раздражение.

Конечно, допустимы и некоторые отклонения от ступенчатого стиля. Например, несколько коротких похожих операторов вполне можно записать в одну строку:

```
a:=0;  b:=0;  c:=0;  f:=4;
```

Этим мы экономим дефицитное место по вертикали экрана или листа бумаги.

5.5. Вложенные операторы if. Сложное условие в операторе if. Логические операции

Согласно синтаксической схеме оператора if, после then и else может стоять любой оператор Паскаля, в том числе и if.

Решим задачу: В компьютер вводится число (пусть для конкретности это будет дальность какого-нибудь выстрела). Если оно находится в интервале от 28 до 30, то напечатать текст ПОПАЛ, иначе - НЕ ПОПАЛ.

Сначала составим алгоритм: Введи число. Если оно больше 28, то надо еще подумать, в противном случае печатай НЕ ПОПАЛ. А о чем же думать? А вот о чем: Если число меньше 30, то печатай ПОПАЛ, иначе печатай НЕ ПОПАЛ.

А теперь по составленному алгоритму напишем программу:

```
VAR a : Real;
BEGIN
   ReadLn (a);
   if a>28   then  if a<30   then WriteLn ('ПОПАЛ')
                             else WriteLn ('НЕ ПОПАЛ')
             else  WriteLn ('НЕ ПОПАЛ')
END.
```

Как компьютер не запутается в этих then и else? Если внимательно присмотреться, то можно увидеть, что двусмысленная ситуация получается только тогда, когда один из if записан без else, а другой - с else. Пример:

d:=3; v:=10; if v<6 then if v<20 then d:=1 else d:=2

Чему будет равняться d после выполнения этого фрагмента - 1, 2 или 3? Ответ зависит от того, какому if принадлежит else - $if\ v<6$ или $if\ v<20$? Чтобы ответить на этот вопрос, пойдем по тексту программы от интересующего нас else к началу и если по пути нам не встретится еще один else. то первый же if, на который мы наткнемся - наш. Если по пути нам встретится еще один else, забудем пока про наш else и найдем сначала if для нового else, а со старым разберемся потом.

Испытайте-ка этот способ в нашем фрагменте и в предыдущей программе. Получается, что else в нашем примере принадлежит $if\ v<20$ и значит, d будет равняться 3.

Задание 25: Усложним задание из 5.2: В компьютер вводятся длины трех отрезков. Компьютер должен ответить на вопрос, правда ли, что эти отрезки могут образовать треугольник.

Указание: Для этого каждый отрезок должен быть меньше суммы длин двух других отрезков.

———————— – ————————

На первый взгляд применение if внутри if создает громоздкую, трудную для понимания программу. Однако, если вы будете аккуратно использовать ступенчатую запись, то программа получится достаточно обозримой.

В Паскале, тем не менее, есть возможность записывать многие программы со вложенными друг в друга if короче и понятнее, используя только один if. Для этого используются логические операции.

Задача "Разборчивая принцесса". В прихожей у принцессы - длинная очередь женихов. Принцессе нравятся только голубоглазые маленького роста. Устав принимать женихов и отбирать из них подходящих, принцесса вместо себя поставила компьютер, написав для него программу, которая говорит ВЫ МНЕ ПОДОЙДЕТЕ тем, у кого цвет глаз голубой и рост меньше 140 см. Остальным программа говорит ДО СВИДАНИЯ.

Вот эта программа:

```
VAR  Tsvet :String;                {Цвет}
     Rost  :Integer;               {Рост}
BEGIN
   WriteLn ('Каков цвет ваших глаз?');
   ReadLn (Tsvet);
   WriteLn ('Введите ваш рост в сантиметрах');
   ReadLn (Rost);
   if  (Tsvet ='Голубой') AND (Rost<140)          {Если цвет голубой  И  рост<140}
       then  WriteLn ('ВЫ МНЕ ПОДОЙДЕТЕ')
       else  WriteLn ('ДО СВИДАНИЯ')
END.
```

Мы видим, что условие в операторе if уже не такое простое, как мы описывали раньше, а сложное, то есть состоящее из двух взятых в скобки условий[6], соединенных знаком **логической операции AND** (читается "энд", переводится "и"). Весь оператор if можно прочесть так - если цвет глаз голубой И рост меньше 140 сантиметров, то печатай ВЫ МНЕ ПОДОЙДЕТЕ, иначе печатай ДО СВИДАНИЯ.

Знак логической операции *AND*, поставленный между двумя условиями, говорит о том, что должны выполняться сразу оба эти условия.

Поэтому наш оператор if ответит ДО СВИДАНИЯ и высоким голубоглазым и высоким неголубоглазым и маленьким неголубоглазым. И лишь маленьким голубоглазым он ответит ВЫ МНЕ ПОДОЙДЕТЕ.

Программа для задачи ПОПАЛ - НЕ ПОПАЛ при использовании логических операций значительно упростится:

[6] В скобки условия нужно брать потому, что «приоритет» операции AND выше, чем у операций сравнения «=» и «<», то есть она выполняется раньше них, точно так же, как, скажем, умножение выполняется раньше сложения. В Паскале все арифметические, логические и другие операции объединены в единую систему приоритетов. Рассмотрение этой системы выходит за рамки книги.

```
VAR a :Real;
BEGIN
   ReadLn (a);
   if (a>28) AND (a<30)  then WriteLn ('ПОПАЛ')
                         else WriteLn ('НЕ ПОПАЛ')
END.
```

Задача "Неразборчивая принцесса". Неразборчивой принцессе нравятся все маленькие независимо от цвета глаз и все голубоглазые независимо от роста. Программа неразборчивой принцессы будет отличаться от программы разборчивой одним единственным знаком логической операции:

if (Tsvet ='Голубой') OR (Rost<140) {Если цвет голубой ИЛИ рост<140}

Знак логической операции OR читается "о", переводится "или".

Поставленный между двумя условиями, знак *OR* говорит о том, что достаточно, если будет выполняться хотя бы одно из них.

Поэтому теперь оператор if ответит ВЫ МНЕ ПОДОЙДЕТЕ и высоким голубоглазым и маленьким голубоглазым и маленьким неголубоглазым. И лишь высоким неголубоглазым он ответит ДО СВИДАНИЯ.

Знаками AND и OR можно объединять сколько угодно условий. Например:

if (a>2) OR (x=b) OR (c<>1) then k:=99 else k:=33

Здесь выполнится оператор $k:=99$, если верно хотя бы одно из трех условий, и лишь когда все три неверны, будет выполняться оператор $k:=33$.

Кроме логических операций AND и OR применяется еще логическая операция **NOT** (читается "нот", переводится "не"). Запись *if NOT(a>b) then...* переводится так - ЕСЛИ НЕПРАВДА, ЧТО a больше b, ТО.... Вот фрагмент:

a:=2; b:=3; if NOT(a>b) then k:=1 else k:=0

Здесь выполнится оператор $k:=1$, так как неправда, что 2>3.

Для простоты примем правило, что условия, к которым применяются логические операции, заключаются в круглые скобки, хотя это не всегда обязательно.

Внутри скобок можно писать не только простые, но и сложные условия. Например, фрагмент *if NOT ((a>2) AND (b>3)) AND (s>8)) then...* можно перевести так - если неправда, что одновременно верны три условия - a>2, b>3, s>8, то...

Примеры:

ФРАГМЕНТ	РЕЗУЛЬТАТ
a:=8; b:=6; **if** (a>b) **AND** (b>7) **then** k:=1 **else** k:=0	k=0
a:=8; b:=6; **if** (a>b) **OR** (b>7) **then** k:=1 **else** k:=0	k=1
a:=8; b:=6; **if** (a<b) **OR** (b>7) **then** k:=1 **else** k:=0	k=0
a:=8; b:=6; **if** **NOT** (a=8) **then** k:=1 **else** k:=0	k=0

Задание 26: "Замысловатая принцесса". Определите, кто нравится принцессе, по фрагменту из ее программы:

```
if    (Tsvet ='Черный')   AND   ((Rost<180)  OR  (Rost>184))
                  then    WriteLn ('ВЫ МНЕ ПОДОЙДЕТЕ')
                  else    WriteLn ('ДО СВИДАНИЯ')
```

До сих пор мы применяли оператор if для выбора одной из двух возможностей. Покажем, как применять его для выбора одной из нескольких. Усложним нашу задачу про ПОПАЛ - НЕ ПОПАЛ:

Человек вводит в компьютер число. Если оно находится в интервале от 28 до 30, то нужно напечатать текст ПОПАЛ, если оно больше или равно 30 - то ПЕРЕЛЕТ, если оно находится на отрезке от 0 до 28, то НЕДОЛЕТ, если число меньше нуля - НЕ БЕЙ ПО СВОИМ.

Вот программа:

```
VAR a : Real;
BEGIN
   ReadLn (a);
   if (a>28) AND (a<30)     then WriteLn ('ПОПАЛ');
   if  a>=30                then WriteLn ('ПЕРЕЛЕТ');
   if (a>=0) AND (a<=28)    then WriteLn ('НЕДОЛЕТ');
   if  a<0                  then WriteLn ('НЕ БЕЙ ПО СВОИМ')
END.
```

Необходимо обращать внимание на аккуратную расстановку знаков сравнения и числовых границ диапазонов. Если во втором if вместо *a>=30* мы напишем *a>=20*, то при вводе числа 25 компьютер даст нам двусмысленный ответ:

```
ПЕРЕЛЕТ
НЕДОЛЕТ
```

Если же мы вместо *a>=30* напишем *a>30*, то при вводе числа 30 мы вообще не получим от компьютера никакого ответа.

Задание 27:

Человек вводит с клавиатуры строку, смысл которой - приветствие при встрече. Компьютер тоже должен ответить приветствием. Отвечать нужно в соответствии со следующей таблицей:

ПРИВЕТСТВИЕ ЧЕЛОВЕКА	ОТВЕТ КОМПЬЮТЕРА
Привет	Привет
Здравствуйте	Здравствуйте
Здорово	Здравствуйте
Добрый день	Салют
Приветик	Салют
Салют	Салют
Здравия желаю	Вольно

5.6. Символьный тип данных Char

Для того, чтобы получить более полное представление о возможностях оператора варианта case, который нам предстоит изучить, познакомимся сначала с новым типом данных. Мы с вами уже познакомились с четырьмя типами данных в Паскале: Integer и LongInt (целые числа), Real (вещественные числа) и String (строки символов). Введем еще один тип - Char (символьный). По-английски Char читается - "кэр", это сокращение от Character - "символ".

Описание *VAR a,b: Char* означает, что переменные *a* и *b* имеют право принимать значения любых символов, с которыми может работать компьютер. (О символах мы говорили в 3.5) Мы можем записать *a:='Л'; b:='+'*, что означает приказ присвоить переменной *a* значение символа *Л*, а переменной *b* - значение символа +. Мы можем записать *ReadLn (a)* , что означает приказ компьютеру ждать ввода с клавиатуры любого одного символа и присвоить его значение переменной *a*. Мы можем записать *WriteLn (b)* и на экране появится плюсик.

Вот программа, переворачивающая любое трехбуквенное слово, введенное человеком с клавиатуры:

```
VAR c1,c2,c3: Char;
BEGIN
   ReadLn (c1,c2,c3);
   WriteLn (c3,c2,c1)
END.
```

Если мы по оператору ReadLn введем символы *ТОК*, то оператор WriteLn напечатает *КОТ*. При вводе нескольких символов одним оператором ReadLn, все символы набираются на клавиатуре подряд, без пробелов, которыми мы привыкли разделять при вводе числовые данные. После ввода последнего символа нажимаем клавишу Enter. Таким образом, ввод трех символов одним оператором ReadLn не отличается от ввода одной трехсимвольной строки. Вообще, тип Char похож на тип String. Строка состоит из символов, да и сам символ - это как бы очень короткая строка длиной в один символ.

5.7. Оператор варианта case

В Паскале есть специальный оператор, который позволяет делать выбор одной из нескольких возможностей. Рассмотрим программу, спрашивающую у ученика его отметку по чистописанию и реагирующую на нее подходящим текстом:

```
VAR  Otmetka: Integer;
BEGIN
   WriteLn ('Какую отметку ты получил по чистописанию?');
   ReadLn (Otmetka);
   CASE otmetka OF     {Перевод: В СЛУЧАЕ ЕСЛИ отметка РАВНА...}
```

```
      1,2    :WriteLn('Кошмар!');
      3      :WriteLn('Неважно');
      4      :WriteLn('Неплохо');
      5      :WriteLn('Молодец!');
      ELSE  WriteLn('Таких отметок не бывает')
   END                          {Конец оператора CASE}
END.
```

Основой программы является **оператор варианта CASE** (читается "кэйс", переводится "случай"). Предлог OF читается "эв". Весь оператор CASE нужно понимать так:

```
В СЛУЧАЕ ЕСЛИ отметка РАВНА
      1 или 2        печатай 'Кошмар!'
      3              печатай 'Неважно'
      4              печатай 'Неплохо'
      5              печатай 'Молодец!'
      ИНАЧЕ          печатай 'Таких отметок не бывает'
КОНЕЦ оператора case
```

В процессе исполнения оператора case компьютер сравнивает значение переменной Otmetka по очереди со всеми значениями, перечисленными перед двоеточиями. Наткнувшись на совпадающее значение, он выполняет оператор, стоящий после двоеточия. На этом исполнение оператора case завершается. Если же совпадающего значения так и не нашлось, то выполняется оператор, стоящий после слова else (в нашей программе он полезен на тот случай, если ученик болен манией величия и вводит число 6). После else может стоять и цепочка операторов, записанных через точку с запятой.

Если вы еще недостаточно хорошо поняли логику оператора case, то обратите внимание на то, что в нашей программе он работает так же, как следующий оператор if:

```
if (Otmetka=1) OR (Otmetka=2)
   then WriteLn('Кошмар!')
   else if Otmetka=3
            then WriteLn('Неважно')
            else if Otmetka=4
                     then WriteLn('Неплохо')
                     else if Otmetka=5
                              then WriteLn('Молодец!')
                              else WriteLn('Таких отметок не бывает')
```

У оператора case есть существенное ограничение - переменная, стоящая после слова case, должна быть так называемого **порядкового типа**. Подробно о порядковых типах мы поговорим в Глава 12, пока же мы знаем три типа, относящиеся к порядковым - это Integer, LongInt и Char. Порядковыми они называются потому, что все их возможные значения можно выстроить по порядку и перечислить с начала до конца: *-32768, ...4,5,6,7,8... 32767* для Integer, *.....'д','е','ж','з','и'....* для Char. Получается, что значения типов Real и String применять в операторе case нельзя, так как совершенно непонятно, как их перечислять по порядку.

Таким образом, задачу про ПОПАЛ-НЕ ПОПАЛ в принципе невозможно решить при помощи case. И этому две причины: 1)переменная имеет тип Real, 2)перед двоеточием в операторе case нельзя писать условия со знаками >, < и т.п.

Вот еще пример программы с оператором case:

```
VAR  a,k : Integer;
BEGIN
   a:=3;
   case  a*a+1  of   {В СЛУЧАЕ ЕСЛИ  a*a+1  РАВНО...}
      8,3,20    :k:=0;
      7,10      :begin k:=1; WriteLn(k) end;
      12 ..18   :k:=3
   end                          {Конец оператора CASE}
END.
```

Эта программа напечатает 1. Здесь мы видим несколько новых для нас элементов:

Во-первых, после слова case стоит не переменная, а выражение, поэтому с перечисленными перед двоеточиями значениями будет сравниваться число 10, полученное как 3*3+1. Кстати, выражение тоже обязано быть порядкового типа (о том, что такое тип выражения, мы строго поговорим в 14.4, а сейчас скажем только, что это выражение, имеющее значения порядкового типа).

Во-вторых, один из операторов, стоящих после двоеточий, составной. Это *begin k:=1; WriteLn(k) end*

В-третьих - конструкция *12 ..18* . Она обозначает то же, что и *12,13,14,15,16,17,18*. Она служит в нашем случае для сокращения записи и называется **диапазоном**.

В-четвертых, здесь отсутствует конструкция *else*. Это значит, что если бы в нашей программе мы вместо *a:=3* написали *a:=0*, то оператор case, не найдя совпадения, не выбрал бы ни один из трех своих вариантов и, не найдя также else, завершил бы свою работу, так ничего и не сделав.

Задание 28:

Ученик вводит с клавиатуры букву русского алфавита. Компьютер должен сказать, какая это буква - гласная, согласная звонкая, согласная глухая или другая какая-нибудь (можно и НЕ ЗНАЮ).

Задание 29: «Калькулятор». Ученик вводит с клавиатуры число, символ арифметического действия (+, -, *, /) и еще одно число. Компьютер должен напечатать результат. *Указание*: Используйте три оператора ReadLn.

Глава 6. Циклические программы

6.1. Оператор перехода GOTO. Цикл. Метки

Цикл – главное средство заставить компьютер много раз сделать одно и то же или похожее. Первое представление о цикле я дал в 2.8. Посмотрим, как осуществить цикл на Паскале. Предположим, мы хотим, чтобы компьютер бесконечно повторял выполнение следующего фрагмента:

```
Write ('Это ');
Write ('тело ');
Write ('цикла');
Write ('      ')
```

в результате чего на мониторе мы бы увидели:

```
Это тело цикла     Это тело цикла     Это тело цикла     Это тело цикла  . . . .
```

Большинство языков программирования (в том числе и Паскаль) устроены так, что операторы выполняются в том порядке, в котором они записаны. Это значит, что после оператора *Write ('Это ')* обязательно выполнится оператор *Write ('тело ')*, а после него - *Write ('цикла')*, а после него - *Write ('   ')*. Все это хорошо. Но нам нужно, чтобы после выполнения оператора *Write ('   ')* Паскаль нарушал этот свой принцип последовательного выполнения операторов и выполнял бы оператор *Write ('Это ')*. Если мы этого добьемся, то дальше все пойдет само собой, так как после *Write ('Это ')* Паскаль автоматически выполнит *Write ('тело ')* и так далее до бесконечности.

Если бы операторы Паскаля можно было писать по-русски, то для достижения нашей цели было бы естественно воспользоваться такой конструкцией:

```
метка m1:     Write ('Это ');
              Write ('тело ');
              Write ('цикла');
              Write ('      ');
              иди к оператору, помеченному меткой m1
```

Здесь мы видим новый для нас "оператор" ИДИ, который выполняется после *Write ('   ')* и единственная работа которого заключается в том, чтобы заставить компьютер перескочить к выполнению оператора *Write ('Это ')*, помеченного меткой *m1*.

А вот как этот фрагмент выглядит реально на Паскале:

```
 m1:   Write ('Это ');
       Write ('тело ');
       Write ('цикла');
       Write ('      ');
       GOTO m1
```

Здесь *GOTO* - оператор перехода, читается " 'гоуту", переводится "иди к", *m1* - метка.

Метка - это произвольное имя или произвольное не слишком большое целое положительное число. Оператор GOTO можно писать в любых местах программы и метку можно ставить перед любым оператором ,заставляя компьютер таким образом перескакивать в программе откуда угодно куда угодно (правда, в сложных программах и внутри сложных операторов эта свобода перескакивания существенно ограничивается. Метка должна отделяться от оператора двоеточием.

Мы пока знаем, что переменная, встречающаяся в программе, должна быть описана выше BEGIN после слова VAR. Метки, встречающиеся в программе, тоже должны быть описаны выше BEGIN после слова **LABEL** (читается "лэйбл", переводится "метка").

Вот наша программа полностью:

```
LABEL m1;
BEGIN
  m1:Write ('Это ');
     Write ('тело ');
     Write ('цикла');
```

```
    Write ('      ');
    goto m1
END.
```

Если вы уже запустили эту программу, то через некоторое время перед вами должен встать жизненно важный вопрос – как же ее остановить? Для этого достаточно прочесть параграф «Выполнение программы» из части IV. Вот вкратце, что вам нужно оттуда знать:

Для прерывания работы программы (в том числе и зациклившейся) существует комбинация клавиш Ctrl-Break *(имеется в виду, что, удерживая нажатой клавишу* Ctrl*, вы должны щелкнуть по клавише* Break*). На экран возвращается окно редактора. Строка программы, на которой она была прервана, выделяется полосой белого цвета. Если вы снова запустите программу, она продолжит работу с прерванного места. Чтобы начать сначала, уберите полосу с экрана клавишами* Ctrl-F2.

Группа операторов, выполняющихся многократно, называется **телом цикла**. У нас это все операторы, начиная с *Write ('Это ')* и кончая *GOTO m1*.

Пример программы:

```
LABEL 8;
VAR  a,k : Integer;
BEGIN
   k:=6;
   a:=100;
   goto 8;
   a:=a+k;
   k:=2*k;
   WriteLn(a);
 8: a:=a+1;
   k:=k+10;
   WriteLn(k,'  ' ,a);
END.
```

Эта программа напечатает 16 101. Операторы выполняются в такой последовательности:

```
k:=6;
a:=100;
goto 8;
a:=a+1;
k:=k+10;
WriteLn(k,'  ' ,a);
```

А операторы *a:=a+k; k:=2*k; WriteLn(a)* выполнены не будут вообще, несмотря на то, что написаны.

Задание 30: Определить без компьютера, что будет печатать программа:

```
LABEL m1,met5;
VAR  n,k : Integer;
BEGIN
      n:=10;
      k:=0;
      WriteLn('Считаем зайцев' );
 met5:Write(n);
      n:=n+k;
      goto m1;
      n:=n+1;
 m1:  Write(' зайцев  ');
      ReadLn;
      k:=k+1;
      goto met5;
      WriteLn('Посчитали зайцев')
END.
```

А теперь, уважаемый читатель, нам с вами пришла пора снова отвлечься от программирования и расширить свои знания о работе на компьютере. Сейчас вам необходимо проделать то, что сказано в части IV в параграфе «Работа с несколькими окнами» и пункте «Работа с окнами пользователя и отладчика» параграфа «Исправление ошибок. Отладка программы».

Задания 31-33:

Написать программы для выполнения следующих заданий:

31) Бесконечно печатать букву А: *AAAAAAAAAA.....*

32) Бесконечно печатать *1000 999 998 997 996 .....*

33) Бесконечно печатать *100 50 25 12.5....* с 8 десятичными знаками.

Примечание: Во всех программах используйте ReadLn для создания паузы на каждом цикле. Чтобы программа шла все-таки довольно быстро, нажмите клавишу ввода и не отпускайте.

6.2. Выход из цикла с помощью if

Интересно рассмотреть применение оператора goto внутри операторов if или case.

Задача: При помощи цикла напечатать на экране:

Начало счета 3 5 7 9 Конец счета

Вот три варианта программы. Первый – самый простой, а второй и третий нам понадобятся в дальнейшем.

1 ВАРИАНТ

```
LABEL m;
VAR   f : Integer;
BEGIN
        Write('Начало счета   ');
        f:=3;
  m:    Write(f,' ');
        f:=f+2;
        if f<=9 then      goto m;

        Write(' Конец счета')
END.
```

2 ВАРИАНТ

```
LABEL m1,m2;
VAR   f : Integer;
BEGIN
        Write('Начало счета   ');
        f:=3;
  m1:   Write(f,' ');
        f:=f+2;
        if f>9    then    goto m2
                  else    goto m1;
  m2:   Write(' Конец счета')
END.
```

Вот в каком порядке выполняются операторы программы первого варианта:

Write('Начало счета ') f:=3 Write(f,' ') **{печатается 3}** f:=f+2 **{f становится равным 5}** if f<=9 goto m Write(f,' ') **{печ. 5}** f:=f+2 **{f = 7}** if f<=9 goto m Write(f,' ') **{печ. 7}** f:=f+2 **{f = 9}** if f<=9 goto m Write(f,' ') **{печ. 9}** f:=f+2 **{f = 11}** if f<=9 Write(' Конец счета')

Здесь оператор goto выполняется три раза. На четвертый раз условие $f<=9$ оказывается ложным и поэтому выполняется не goto, а следующий за if оператор *Write(' Конец счета')*, то есть программа выходит из цикла и завершает свою работу.

3 ВАРИАНТ

```
LABEL m1,m2,m3;
VAR   f : Integer;
BEGIN
        Write('Начало счета   ');
        f:=3;
  m1:   if f<=9  then     goto m3
                  else    goto m2;
  m3:   Write(f,' ');
        f:=f+2;
        goto m1;
  m2:   Write(' Конец счета')
END.
```

Задания 34-36:

34) Напечатать *1 2 3 4 . . . 99 100 99 . . . 3 2 1.*

35) "*Таблицы Брадиса*"- вычислить и напечатать с 6 десятичными знаками квадраты чисел *0.000 0.001 0.002 0.003 . . . 0.999 1.000.*

36) Для x=2700, 900, 300, 100 . . . и т.д. вычислять и печатать $y=x/4 + 20$ и $z=2y+0.23$ до тех пор, пока yz не станет меньше $1/x$.

Совет: Теперь, когда вы владеете отладочным режимом, смело применяйте его всякий раз, когда ваша программа не хочет делать то, что нужно. Зачем ломать голову над непослушной программой? – Берегите серое вещество, жмите F7!

6.3. Оператор цикла repeat

Циклы настолько широко применяются в программах, что у программистов давным-давно появилась потребность написать специальный оператор цикла, не использующий оператор goto, так как последний неудобен хотя бы тем, что у программистов, пишущих большие программы, много времени и внимания уходит на поиск взглядом меток в тексте программы.

В Паскале - три оператора цикла: repeat, while и for. Первым изучим оператор repeat.

Конструкция **repeat . . . . . until a+2>3*b** читается "ри'пит.....ан'тил...", а переводится "повторяй.....до тех пор, пока a+2 не станет больше 3*b".

Составим с использованием оператора repeat программу решения задачи о печати чисел $3\ \ 5\ \ 7\ \ 9$ из предыдущего параграфа. Для того, чтобы точно определить работу оператора repeat, приведем ее параллельно со вторым вариантом программы решения этой задачи из того же параграфа:

2 ВАРИАНТ	ВАРИАНТ С REPEAT

LABEL m1,m2;
VAR f : Integer;
BEGIN
Write('Начало счета ');
f:=3;
m1: Write(f,' ');
f:=f+2;
if f>9 **then goto** m2
else goto m1;
m2: Write(' Конец счета')
END.
	VAR f : Integer;
	BEGIN
	Write('Начало счета ');
	f:=3;
	repeat
	Write(f,' ');
	f:=f+2;
	until f>9;
	Write(' Конец счета')
	END.

Порядок работы обеих программ совершенно одинаков, так что можно считать слово *repeat* заменой метки *m1*, а конструкцию *until f>9* считать заменой оператора *if f>9 then goto m2 else goto m1*.

Синтаксис оператора repeat:

REPEAT оператор **;** оператор **;**. . . **;** оператор **UNTIL** условие

Вкратце работу оператора repeat можно описать так: Повторяй выполнение операторов, стоящих между словами repeat и until, до тех пор, пока не станет истинным условие.

Более подробно работа оператора repeat описывается так:

Сначала компьютер по очереди выполняет операторы, стоящие после слова *repeat*, пока не дойдет до слова *until*, после чего проверяет истинность условия, стоящего после *until*. Если условие ложно, то компьютер снова по очереди выполняет эти операторы и снова проверяет истинность условия и т.д. Если условие оказывается истинным, то работа оператора *repeat* прекращается и компьютер переходит к выполнению следующего по порядку оператора.

Задача: Компьютер предлагает человеку ввести слово, после чего распечатывает это слово, снабдив его восклицательным знаком. Затем снова предлагает ввести слово и так до тех пор, пока человек не введет слово "Хватит". Распечатав его с восклицательным знаком, компьютер отвечает "Хватит так хватит" и заканчивает работу.

Придумаем строковую переменную, в которую человек будет с клавиатуры вводить слово. Назовем ее *Slovo*.

```
VAR Slovo : String;
BEGIN
   repeat
```

```
    WriteLn('Введите слово');
    ReadLn(Slovo);
    WriteLn(Slovo, '!')
  until Slovo='Хватит';
  WriteLn('Хватит так хватит')
END.
```

Задание 37: Усложним эту задачу. Пусть компьютер перед распечаткой каждого слова ставит его порядковый номер.

Задание 38-39: Выполнить с применением оператора repeat последние два задания из предыдущего параграфа.

Задание 40: Если камень бросить горизонтально со 100-метровой башни со скоростью v=20м/с, то его расстояние от башни по горизонтали (s) будет выражаться формулой s=vt, где t – время полета камня в секундах. Высота над землей h будет выражаться формулой $h=100 - 9.81t^2/2$. Вычислять и печатать t, s и h для t=0, 0.2, 0.4, 0.6 и так далее до тех пор, пока камень не упадет на землю.

6.4. Оператор цикла while

Синтаксис оператора while:

```
WHILE условие DO оператор
```

Слово **while** читается "вайл", слово **do** - "ду", вся конструкция переводится так - **Пока** условие истинно, **делай** оператор. Например, *while a>b do b:=b+1*.

Работает оператор while так:

Сначала компьютер проверяет истинность условия, стоящего после слова *while*. Если условие истинно, то выполняется оператор, стоящий после *do*. Затем снова проверяется истинность условия и в случае истинности снова выполняется этот оператор. И т.д. Если условие ложно, то оператор *while* прекращает свою работу и компьютер переходит к выполнению следующего оператора.

Оператор, стоящий после while, вполне может быть составным, поэтому тело цикла у оператора while, так же как и у оператора repeat, может состоять из многих операторов.

Решим при помощи while ту же задачу о печати чисел *3 5 7 9*, что в предыдущем параграфе решили с помощью repeat. Для того, чтобы точно определить работу оператора while, приведем программу ее решения параллельно с третьим вариантом программы из 6.2:

3 ВАРИАНТ	ВАРИАНТ С WHILE

LABEL m1,m2,m3;
VAR f : Integer;
BEGIN
Write('Начало счета ');
f:=3;
m1: **if** f<=9 **then** **goto** m3
else **goto** m2;
m3: Write(f,' ');
f:=f+2;
goto m1;
m2: Write(' Конец счета')
END.
	VAR f : Integer;
	BEGIN
	Write('Начало счета ');
	f:=3;
	while f<=9 **do**
	begin
	Write(f,' ');
	f:=f+2;
	end;
	Write(' Конец счета')
	END.

Как видите, здесь после *do* стоит составной оператор *begin Write(f,' '); f:=f+2; end*. Последовательность исполнения операторов и проверки условий в обеих программах совершенно аналогичны.

Типичная ошибка начинающих – небрежное обращение со знаками сравнения. Многие не видят большой разницы в том, как записать – *while f<=9* или *while f<9* , а затем, «недополучив» результат, удивляются, почему. И здесь лучшим средством для понимания является отладочный режим. Попробуйте ошибочный вариант последней программы (с *while f<9*) выполнить в пошаговом режиме с использованием окон пользователя и отладчика. Для этого введите в окно отладчика две вещи: переменную *f* и выражение *f<9* (оно может иметь только два значения: *true* - *истина* и *false* - *ложь*, другими словами – «условие выполнено» и «условие не выполнено»).

Задание 41: Вычислять с использованием while квадратные корни из чисел *900, 893, 886, 879* и т.д. до тех пор, пока это можно делать.

6.5. Отличия операторов repeat и while

Отличий три:

- Компьютер выходит из цикла оператора *repeat* тогда, когда условие истинно, а из цикла оператора *while* - когда условие ложно.
- *while* может ни разу не выполнить оператор, стоящий после *do*. *repeat* же хотя бы раз операторы, стоящие между *repeat* и *until*, выполнит.

  Так, фрагмент *k:=8; repeat k:=1 until 3>2; WriteLn(k)* напечатает 1.
  А фрагмент *k:=8; while 2>3 do k:=1; WriteLn(k)* напечатает 8.
- При компиляции оператор while дает несколько более эффективную программу, чем оператор repeat.

Часто эти отличия для начинающих малосущественны, поэтому выбирайте оператор по вкусу. Мне, например, надоели паскалевские begin и end, поэтому я охотнее пользуюсь оператором repeat.

6.6. Оператор цикла for

Выполняя программу печати чисел *3 5 7 9*, оператор repeat выполнил цикл 4 раза. То же самое сделал и оператор while. Однако, обычно, когда мы пишем операторы repeat и while, нам совсем неважно знать, сколько раз они выполнят цикл. Тем не менее, существует много задач, для решения которых цикл нужно выполнить именно определенное количество раз. В этом случае удобно использовать оператор цикла for.

Задача: 200 раз напечатать слово ФУТБОЛ.

Попробуем сначала решить задачу при помощи оператора goto. Начнем с такого фрагмента:

```
metka:  WriteLn('ФУТБОЛ');
        goto metka
```

Но здесь цикл будет повторяться бесконечно, а нам нужно только 200 раз. Мы уже видели, что для выхода из цикла оператор goto нужно включить в состав оператора if. Кроме этого нужна переменная, меняющая свое значение от одного выполнения цикла к следующему. Придумаем этой величине какое-нибудь имя, скажем *i*. Проще всего задачу решает такой фрагмент:

```
        i:=1;
metka:  WriteLn('ФУТБОЛ');
        i:=i+1;                    {увеличение i на 1}
        if  i<=200 then goto metka
END.
```

Здесь *i* вначале равно 1, но к каждому следующему выполнению цикла оно увеличивается на 1. В первый раз выполняя оператор if, компьютер проверяет условие 2<=200 и найдя его истинным, выполняет оператор *goto metka*. Во второй раз проверяется условие 3<=200 и т.д. В 199-й раз компьютер проверяет условие 200<=200 и найдя его истинным, выполняет оператор *goto metka*. В 200-й раз компьютер проверяет условие 201<=200 и найдя его ложным, выходит из цикла.

В нашем фрагменте "полезную" работу выполняет только одна строка из четырех - *WriteLn('ФУТБОЛ')*. Остальные три строки заняты тем, что обеспечивают выполнение "полезной" строки ровно 200 раз. Нам пришлось организовать специальную переменную, значение которой в каждый момент выполнения программы говорит о том, в какой раз выполняется цикл. Переменная с таким свойством называется **счетчиком циклов**.

А теперь запишем программу полностью, правда, несколько усложнив ее, так чтобы логика ее выполнения полностью соответствовала логике выполнения программы с оператором for, которую я привожу параллельно и поясняю немедленно.

```
LABEL          m1,m2;
VAR    i : Integer;
BEGIN  i:=1;
   m1:  if  i>200 then goto m2;
        WriteLn('ФУТБОЛ');
        i:=i+1;
        goto m1;
    m2:
END.
```

```
VAR i : Integer;
BEGIN
        for i:=1 to 200 do
                WriteLn('ФУТБОЛ')


END.
```

Слово **for** читается "фо", переводится "для". Слово **to** читается "ту", переводится "до". Слово **do** читается "ду", переводится "делай". Конструкция **for i:=1 to 200 do** по-русски читается так: *Для i, изменяющегося от 1 до 200, делай оператор, стоящий после слова do*. Смысл повторения здесь такой же, как и в операторе while. Оператор, стоящий после do, тоже, конечно, может быть составным.

Синтаксис оператора for:

FOR имя **:=** выражение1 **TO** выражение2 **DO** оператор

Пример записи: *for j:=a+b to 2*s do k:=k+1*.

Пояснения к синтаксической схеме:

имя - это имя произвольной переменной порядкового типа (см. 5.7 и 12.8), в частности целочисленной, называемой **переменной цикла**,

выражение1 и *выражение2* - произвольные выражения порядкового типа, в частности - целого.

Работа оператора for:

Прежде всего вычисляется *выражение1*, и переменной цикла (пусть это будет *i*) присваивается его значение. Затем вычисляется *выражение2* и сравнивается с *i*. Если *i* > *выражения2*, то оператор *for* завершает свою работу, так ничего и не сделав. В противном случае выполняется оператор, стоящий после *do*. После выполнения этого оператора значение *i* увеличивается на *1* и снова сравнивается с *выражением2*. Если *i* > *выражения2*, то оператор *for* завершает свою работу, иначе снова выполняется оператор, стоящий после *do*, снова *i* увеличивается на 1 и т.д.

В нашем примере переменная *i* кроме того, что обеспечивает нам выход из цикла, никакой "полезной" работы не выполняет. Усложним же немного задачу. Пусть компьютер печатает такую информацию:

10 ФУТБОЛ 11 ФУТБОЛ 12 ФУТБОЛ . . . . . 200 ФУТБОЛ

Вот программа:

```
VAR i : Integer;
BEGIN
   for i:=10 to 200 do
      begin     Write(i);
                Write(' ФУТБОЛ  ')
      end
END.
```

Здесь после do стоит уже составной оператор.

Можно ли удобно использовать оператор for для печати такой информации?:

100 ФУТБОЛ 99 ФУТБОЛ 98 ФУТБОЛ . . . . . 40 ФУТБОЛ

Вполне, так как оператор for позволяет не только увеличивать, но и уменьшать переменную цикла. Однако, для этого нельзя писать *for i:=100 to 40*, а нужно писать *for i:=100* ***downto*** *40*. Читается *downto* - "*даунту*", переводится буквально "*вниз до*". Соответственно, для выхода из цикла должно быть истинным не условие *i* > *выражения2*, а условие *i* < *выражения2*.

Вот объединенный синтаксис оператора for:

FOR имя **:=** выражение1 **TO** | **DOWNTO** выражение2 **DO** оператор

Вертикальная черта | между двумя элементами конструкции «TO и DOWNTO» говорит о том, что в конструкции должен присутствовать один из этих элементов.

Задание 42: Напечатать с помощью оператора for:

Прямой счет: -5 -4 -3 -2 -1 0 1 2 3 4 5 Обратный счет: 5 4 3 2 1 0 -1 -2 -3 -4 -5 Конец счета

Глава 7. Типичные маленькие программы

Каждому программисту известны такие понятия, как счетчик, сумматор, вложенные циклы и подобные понятия, составляющие элементарную технику программирования. Без них не обходится ни одна реальная программа. В этой главе я не буду вводить новых операторов, а покажу, как программировать типичные задачи, в том числе и те, что используют упомянутые понятия.

7.1. Вычислительная циклическая программа

Задача: Во дворце 40 залов. Известны длина, ширина и высота каждого зала. Вычислить площадь пола и объем каждого зала.

Сначала напишем фрагмент для одного зала:

```
ReadLn (dlina, shirina, visota);
S:=dlina*shirina;                    {Площадь пола}
V:=S*visota;                         {Объем}
WriteLn(S,'  ',V)
```

Для решения задачи этот фрагмент нужно выполнить 40 раз, для чего вполне естественно использовать оператор for:

```
VAR i, dlina, shirina, visota, S, V: Integer;
BEGIN
   for i:=1 to 40 do begin
      ReadLn (dlina, shirina, visota);
      S:=dlina*shirina;
      V:=S*visota;
      WriteLn(S,'  ',V)
   end {for}
END.
```

Обратите внимание, что здесь мы несколько модифицировали описанный нами в 5.4 ступенчатый стиль, а именно записали *end* не под соответствующим ему *begin*, а под соответствующим ему *for*. Эта практика также распространена, так как экономит место по вертикали. Мне она нравится больше, поэтому я буду ее придерживаться. Чтобы не спутаться, откуда взялся *end*, пишем рядом комментарий *{for}*.

Теперь создадим более дружественный интерфейс, для чего, кроме всего прочего, дадим возможность пользователю самому задавать число залов во дворце:

```
VAR   i, dlina, shirina, visota, N, S, V : Integer;
BEGIN
   WriteLn('Введите число залов');
   ReadLn (N);              {N - число залов}
   for i:=1 to N do begin
      WriteLn('Введите длину,ширину и высоту зала');
      ReadLn (dlina, shirina, visota);
      S:=dlina*shirina;
      V:=S*visota;
      WriteLn('Площадь пола=',S,' Объем зала=',V)
   end
END.
```

Здесь курсивом я обозначил новые по сравнению с предыдущей программой элементы.

Пусть во дворце три зала размерами 20*15*4, 30*20*5 и 10*5*3. В этом случае мы вводим N=3 и оператор for выполняет цикл три раза. На каждом выполнении цикла компьютер останавливается на операторе *ReadLn (dlina, shirina, visota)*, мы вводим числа и получаем результаты:

```
Площадь пола=300 Объем зала=1200
Площадь пола=600 Объем зала=3000
Площадь пола=50 Объем зала=150
```

Задание 43: Даны стороны N кубиков. Вычислить объем каждого.

7.2. Роль ошибок

Из 2.2 мы знаем, что по ошибочной программе компьютер выдает ошибочные результаты. Например, если в нашей программе мы вместо *V:=S*visota* напишем *V:=S+visota*, то результаты будут такими:

```
Площадь пола=300 Объем зала=304
Площадь пола=600 Объем зала=605
Площадь пола=50 Объем зала=53
```

Если случайно вместо *for i:=1 to N* написать *for i:=2 to N* то результаты будут такими:

```
Площадь пола=300 Объем зала=1200
Площадь пола=600 Объем зала=3000
```

На этом программа закончит работу и не спросит размеров третьего зала. Вам не кажется странным, что она посчитала 1 и 2 залы, а не 2 и 3? Если кажется, то учтите, что пользователь ничего не знает об ошибке в программе, а компьютер не говорит ему, размеры какого по счету зала ему нужно вводить.

Задания 44-45:

Определите без компьютера, что будет, если

44) строку *for i:=1 to N do begin* поместить под строкой *ReadLn (dlina, shirina, visota)*

45) поменять местами строки *WriteLn('Площадь пола=',S,' Объем зала=',V)* и *end*

Если задания не получаются, введите программы в компьютер и используйте отладочный режим.

7.3. Счетчики

Задача 1: В компьютер с клавиатуры вводятся числа. Компьютер после ввода каждого числа должен печатать, сколько среди них уже введено положительных.

Фрагмент, решающий задачу:

```
    c:=0;                          {Обнуляем счетчик}
m:  ReadLn(a);                     {Вводим очередное число}
    if a>0 then c:=c+1;
    WriteLn('Из них положительных - ' ,c);
    goto m
```

Пояснения: В 6.6 мы придумали переменную *i*, которую назвали счетчиком циклов. Здесь мы тоже придумали переменную *c*. Она у нас выполняет роль **счетчика** положительных чисел. Сердце счетчика - оператор *c:=c+1*. Именно он в нужный момент увеличивает счетчик на 1. Но и без *if a>0 then* тоже никак нельзя. Если бы его не было, то *c* подсчитывал бы все числа без разбору, то есть был бы обыкновенным счетчиком циклов. В нашем же фрагменте увеличение *c* на 1 выполняется не всегда, а лишь при положительном *a*.

Пусть мы вводим числа 8, -2, 10 . . . В этом случае порядок выполнения операторов будет такой:

Оператор	a	c	Печать
c:=0	?	0	
ReadLn(a)	8	0	
if a>0 then c:=c+1	8	1	
WriteLn('Из них положительных - ' ,c)	8	1	Из них положительных 1
goto m	8	1	
ReadLn(a)	-2	1	
if a>0 then c:=c+1	-2	1	
WriteLn('Из них положительных - ' ,c)	-2	1	Из них положительных 1
goto m	-2	1	
ReadLn(a)	10	1	
if a>0 then c:=c+1	10	2	
WriteLn('Из них положительных - ' ,c)	10	2	Из них положительных 2
goto m	10	2	

Не забывайте обнулять счетчик перед входом в цикл, а не то он начнет считать вам не с нуля, а бог знает с чего. Как бы вам понравилось, если бы таксист в начале поездки не обнулил счетчик?

В нашем фрагменте значения счетчика печатаются при каждом выполнении цикла. Изменим задачу.

Задача 2: В компьютер вводится ровно 200 чисел. Компьютер должен подсчитать и один раз напечатать, сколько среди них положительных.

Программа:

```
VAR  c,i :Integer;
     a   :Real;
BEGIN
   c:=0;                        {Обнуляем счетчик}
   for i:=1 to 200 do begin
      ReadLn(a);
      if a>0 then c:=c+1
   end {for};
   WriteLn('Из них положительных - ' ,c)
END.
```

Пояснения: Путь рассуждений здесь тот же, что и в первой задаче. В результате применения оператора for фрагмент *ReadLn(a);if a>0 then c:=c+1* выполняется ровно 200 раз, благодаря чему счетчик *c* накапливает нужное значение. Оператор WriteLn выполняется только один раз и печатает это значение.

Совет: Если вы запускаете эту программу в компьютере, то с числом 200 возиться крайне долго. Поменяйте его на 3 или 4. Смысл программы от этого не изменится.

Задание 46: Что будет, если

1) Вместо *c:=0* написать *c:=10.*
2) Вместо *c:=c+1* написать *c:=c+2*.
3) Строки *end {for}* и *WriteLn* поменять местами.
4) Строки *c:=0* и *for* поменять местами.
5) Строки *for* и *ReadLn* поменять местами.

Задача 3: В компьютер один за другим вводятся произвольные символы. Ввод заканчивается символом "**/**". Подсчитать, какой процент от общего числа введенных символов составляют символ "**W**" и символ "**:**" по отдельности.

Здесь мы организуем три счетчика одновременно: *cW* и *cDv* - для подсчета букв *W* и двоеточий соответственно, а также *i* - счетчик циклов, то есть общего числа введенных символов.

Программа:

```
VAR  i,cW,cDv,procent_W, procent_Dv  : Integer;
     simvol                          :Char;
BEGIN
   i:=0;  cW:=0;  cDv:=0;            {Обнуляем все три счетчика}
   repeat                            {Повторяй цикл}
      ReadLn (simvol);                  {Введи символ}
      i:=i+1;                           {«Посчитай» его}
      case simvol of
         'W':cW:=cW+1;                  {Если это W, увеличь счетчик символов W}
         ':'  :cDv:=cDv+1               {Если это :, увеличь счетчик символов :}
      end
   until simvol = '/';               {пока не наткнешься на символ /}
   procent_W      :=Round(100*cW/i);  {Вычисляй процент символов W}
   procent_Dv     :=Round(100*cDv/i); {Вычисляй процент символов :}
   WriteLn(procent_W,'  ',procent_Dv)
END.
```

Задание 47: В компьютер вводится N чисел. Подсчитать по отдельности количество отрицательных, положительных и тех, что превышают число 10.

Задание 48: В компьютер вводятся пары целых чисел. Подсчитать, сколько среди них пар, дающих в сумме число 13. Подсчет закончить после ввода пары нулей.

Напомню, что пару чисел можно ввести оператором *ReadLn(a,b)*.

7.4. Сумматоры

Если вы поняли идею счетчика, то понять идею сумматора вам будет нетрудно. Посмотрим, как будет работать следующий фрагмент:

```
        s:=0;                   {Обнуляем сумматор. Это не менее важно, чем обнулить счетчик}
m:      ReadLn(a);
        s:=s+a;                 {Увеличиваем сумматор}
        WriteLn('Сумма=', s);
        goto m;
```

Пусть мы вводим числа 8, 4, 10 . . . В этом случае порядок выполнения операторов будет такой:

Оператор	a	s	Печать
s:=0	?	0	
ReadLn(a)	8	0	
s:=s+a	8	8	
WriteLn('Сумма=' ,s)	8	8	Сумма=8
goto m	8	8	
ReadLn(a)	4	8	
s:=s+a	4	12	
WriteLn('Сумма=' ,s)	4	12	Сумма=12
goto m	4	12	
ReadLn(a)	10	12	
s:=s+a	10	22	
WriteLn('Сумма=' ,s)	10	22	Сумма=22
goto m	10	22	
…….			

Как видите, в ячейке s накапливается сумма вводимых чисел a, поэтому назовем эту ячейку **сумматором**. Отличие сумматора от счетчика в том, что счетчик увеличивается на 1 оператором $c:=c+1$, а сумматор - на суммируемое число оператором $s:=s+a$.

Задача: В компьютер вводится N чисел. Вычислить и один раз напечатать их сумму.

Программа:

```
VAR  i,N    :Integer;
     a,s    :Real;
BEGIN
   ReadLn(N);
   s:=0;
   for i:=1 to N do begin
      ReadLn(a);
      s:=s+a
   end {for};
   WriteLn('Сумма равна ' ,s:20:10)
END.
```

Задание 49: Пусть N=2, a=5 и 3. Тогда по этой программе Паскаль напечатает 8. Что он напечатает, если:

1) Вместо *s:=0* написать *s:=10.*
2) Вместо *s:=s+a* написать *s:=s+a+1*.
3) Строки *end {for}* и *WriteLn* поменять местами.
4) Строки *s:=0* и *for* поменять местами.
5) Строки *for* и *ReadLn* поменять местами.
6) Строки *s:=s+a* и *end {for}* поменять местами.
7) Вместо *for i:=1 to N* написать *for i:=2 to N.*

Задания 50-52: Написать программы для следующих задач:

50) Во дворце 40 залов. Известны длина и ширина каждого зала. Вычислить площадь пола всего дворца.
51) Вычислить средний балл учеников вашего класса по физике.
52) Вычислить произведение N произвольных чисел.

7.5. Вложение циклов в разветвления и наоборот

Реальная программа на Паскале представляет собой сложную мозаику из циклических и разветвляющихся частей, вложенных друг в друга. Мы уже видели в 5.7, как в оператор case был вложен оператор for. В свою очередь в оператор цикла могут быть вложены другие операторы, как в 7.3, и так до бесконечности.

Для тренировки определите, что напечатает следующий фрагмент:

```
for i:=1 to 5 do begin
    a:=9;
    if i*i  = a  then   for k:=5 to 8 do Write(k)
                 else   WriteLn(1997);
end {for}
```

Ответ:

```
1997
1997
56781997
1997
```

7.6. Вложенные циклы

Поставим себе задачу - напечатать таблицу умножения. В следующем виде:

```
1*1=  1 1*2=  2 1*3=  3 1*4=  4 1*5=  5 1*6=  6 1*7=  7 1*8=  8 1*9=  9
2*1=  2 2*2=  4 2*3=  6 2*4=  8 2*5= 10 2*6= 12 2*7= 14 2*8= 16 2*9= 18
3*1=  3 3*2=  6 3*3=  9 3*4= 12 3*5= 15 3*6= 18 3*7= 21 3*8= 24 3*9= 27
4*1=  4 4*2=  8 4*3= 12 4*4= 16 4*5= 20 4*6= 24 4*7= 28 4*8= 32 4*9= 36
5*1=  5 5*2= 10 5*3= 15 5*4= 20 5*5= 25 5*6= 30 5*7= 35 5*8= 40 5*9= 45
6*1=  6 6*2= 12 6*3= 18 6*4= 24 6*5= 30 6*6= 36 6*7= 42 6*8= 48 6*9= 54
7*1=  7 7*2= 14 7*3= 21 7*4= 28 7*5= 35 7*6= 42 7*7= 49 7*8= 56 7*9= 63
8*1=  8 8*2= 16 8*3= 24 8*4= 32 8*5= 40 8*6= 48 8*7= 56 8*8= 64 8*9= 72
9*1=  9 9*2= 18 9*3= 27 9*4= 36 9*5= 45 9*6= 54 9*7= 63 9*8= 72 9*9= 81
```

Начнем с малого - пусть нужно напечатать

```
1*1=1
```

Вот фрагмент программы:

Фрагмент 1

```
a:=1;
b:=1;
proizv:=a*b;
Write(a,  '*'  ,b,  '='  ,proizv)
```

Здесь в операторе Write 5 элементов:

* сомножитель a,
* символ знака умножения '*',
* сомножитель b,
* символ '=',
* значение произведения proizv

Усложним задачу. Попробуем заставить компьютер напечатать первую строку таблицы:

```
1*1=  1 1*2=  2 1*3=  3 1*4=  4 1*5=  5 1*6=  6 1*7=  7 1*8=  8 1*9=  9
```

Замечаем, что здесь нам нужно решить 9 элементарных задач на вычисление произведения, первую из которых решает фрагмент 1. Все они очень похожи и различаются лишь значением второго сомножителя. Таким образом, для решения каждой из 9 задач подошел бы наш фрагмент 1, если бы в нем в операторе *b:=1* вместо единицы стояла нужная цифра. В данном случае идеально подходит оператор for:

Фрагмент 2

```
a:=1;
for b:=1 to 9 do begin
    proizv:=a*b;
    Write(a,  '*'  ,b,  '='  ,proizv,  ' ' )
end {for}
```

Для того, чтобы печать была аккуратной, оператор Write мы дополнили символом пробела ' '. Он нужен для того, чтобы отдельные столбцы таблицы не сливались.

Следующая ступень усложнения - последняя - напечатать не одну строку таблицы, а девять. Для этого фрагмент 2 должен быть выполнен 9 раз, каждый раз - с новым значением a. Чтобы этого достичь, "обнимем" фрагмент 2 оператором for точно так же, как мы это сделали с фрагментом 1.

Фрагмент 3

```
for a:=1 to 9 do
   for b:=1 to 9 do begin
      proizv:=a*b;
      Write(a,  '*'  ,b,  '='  ,proizv,  ' ' )
   end {for b}
end {for a}
```

Печатать фрагмент 3 будет неаккуратно. Приведем окончательную запись программы с необходимыми добавлениями для аккуратной печати, а также для удобства объяснений снабдим программу комментариями с нумерацией строк:

```
VAR a,b,proizv: Integer;                          {1}
BEGIN                                             {2}
   for a:=1 to 9 do begin                         {3}
      WriteLn;                                    {4}
      for b:=1 to 9 do begin                      {5}
         proizv:=a*b;                             {6}
         Write(a,  '*'  ,b,  '='  ,proizv:3,  ' ' )   {7}
      end {for b}                                 {8}
   end {for a}                                    {9}
END.                                              {10}
```

WriteLn нужен для того, чтобы каждая новая строка таблицы начиналась с новой строки экрана.

Формат *:3* означает, что на изображение произведения на экране отведено три позиции. Формат в нашем примере нужен для того, чтобы разные по количеству цифр произведения (например, 4 и 25) занимали на экране одинаковое по размеру место, а то не получится у нас аккуратных столбиков в таблице.

В целом программа иллюстрирует идею вложенных циклов, когда один, внутренний, цикл вложен внутрь другого, внешнего. У нас тело внешнего цикла (строки 4 и 5) выполняется 9 раз, а тело внутреннего (строки 6, 7 и 8) - 81 раз, так как на каждое выполнение строки 5 оно выполняется 9 раз.

Задания 53-56:

53) Распечатать все возможные сочетания из двух цифр - первая цифра может быть любой от 3 до 8, вторая - любой от 0 до 7. Например, 36, 44, 80.

54) Распечатать все возможные сочетания из четырех цифр, каждая из которых может принимать значения 1,2,3. Например, 2123, 3312, 1111.

55) Подсчитать количество таких сочетаний.

56) Подсчитать количество неубывающих сочетаний, то есть таких, где каждая следующая цифра не меньше предыдущей - 1123, 1223, 2222 и т.п., но не 3322.

7.7. Поиск максимального из чисел

Задача программисту: Найти максимальное из вводимых в компьютер чисел.

Задача рыбаку: Принести домой самую большую из выловленных рыб.

Решение рыбака: Рыбак приготовил для самой большой рыбы пустое ведро. Первую пойманную рыбу рыбак не глядя бросает в это ведро. Каждую следующую рыбу он сравнивает с той, что в ведре. Если она больше, то он бросает ее в ведро, а ту, что была там раньше, выпускает в реку.

Решение программиста: Программист приготовил для самого большого числа ячейку и придумал ей название, скажем, max. Первое число программист не глядя вводит в эту ячейку. Каждое следующее число (назовем его chislo) он сравнивает с max. Если оно больше, то он присваивает переменной max значение этого числа.

Напишем программу для определения максимального из 10 вводимых чисел:

```
VAR  i, chislo, max  :Integer;
BEGIN
   ReadLn(max);                    {первую рыбу - в ведро}
   for i:=2 to 10 do begin         {ловим остальных рыб:}
```

```
    ReadLn(chislo);                  {поймали очередную рыбу}
    if chislo>max then max:=chislo   {и если она больше той, что в ведре, бросаем ее в ведро }
  end {for};
  WriteLn(max)                       {несем самую большую рыбу домой}
END.
```

Задание 57: Найти из N чисел минимальное. Каким по порядку было введено минимальное число? *Указание*: для номера минимального числа тоже нужно отвести специальную ячейку.

Задание 58: У вас есть данные о росте ваших одноклассников. Правда ли, что рост самого высокого отличается от роста самого низкого больше, чем на 40 см.?

Глава 8. Процедуры

Смысл и выгода процедур вам известны из 2.8. Напомню, что процедуры нужны для того, чтобы программа была короче, и чтобы ее было легче прочесть. Ни одна профессиональная программа не обходится без процедур или без их старших братьев - объектов, рассмотрение которых выходит за рамки начального курса.

Я мог бы объяснить вам процедуры, не вводя новых операторов, однако мне кажется, что лучше всего их объяснять на музыкальном примере, поэтому я предварительно расскажу вам, как заставить компьютер звучать и исполнять мелодии.

8.1. Компьютер звучит

Если даже в вашем компьютере нет звуковой карты, все равно он может звучать. Посмотрим, что заставляет его сделать такая программа:

```
USES CRT;
BEGIN
   Sound(300)
END.
```

Пояснения:

Если мы хотим, чтобы наш компьютер настроился на работу со звуком, мы должны первой строкой программы написать *Uses CRT*. Подробно о том, что это значит, я расскажу в 9.1, а сейчас не будем отвлекаться.

Единственный оператор программы *Sound(300)* приказывает компьютеру включить ровный однообразный звук частотой 300 колебаний в секунду (герц). Слово Sound звучит "'саунд", переводится "звук". Для тех, кто не знает, поясню, что частота определяет высоту звука. *Sound(300)* - это звук средней высоты . *Sound(6000)* - это звук высокий, тонкий, как комариный писк. *Sound(40)* - звук низкий, толстый.

Итак, все действие нашей программы заключается в том, что включается звук. А что дальше? Когда он выключается? А никогда! Программа, выполнившись мгновенно, прекращает свою работу, и мы остаемся один на один со звуком. Через две-три минуты он начинает нам надоедать. Пытаясь его прекратить, мы выходим из среды Паскаля - не помогает. В общем, звук продолжается все то время, пока компьютер включен. В остальном он никак не мешает компьютеру правильно работать. Мы можем запустить другую программу - звук будет сопровождать нас. Самый простой способ избавиться от звука - перезапустить компьютер. Другой способ - выполнить программу, в которой есть оператор NoSound:

```
USES CRT;
BEGIN
   NoSound
END.
```

Оператор *NoSound* (звучит "ноу 'саунд", переводится "нет звука"), выключает звук. Совет: Работая с любой звуковой программой, откройте еще одно окно и введите туда эту программу с *NoSound*. Не пожалеете!

Теперь рассмотрим такую программу:

```
USES CRT;
BEGIN
   Sound (300);  Delay (2000);  NoSound
END.
```

Здесь мы видим новый для нас оператор *Delay (2000)*. Он читается "ди'лэй", переводится "отсрочка" или "пауза". Его действие в том, что он приостанавливает работу программы на 2000 миллисекунд или, что то же самое, на 2 секунды. *Delay (1000)* приостанавливает работу программы на 1 секунду, *Delay (500)* - на полсекунды и т.д. (Должен сказать, что на самом деле продолжительность паузы сильно зависит от быстродействия компьютера).

Итак, оператор *Sound (300)* включает звук. Сразу после этого оператор *Delay (2000)* приостанавливает работу программы на 2 сек. Но звук этот оператор не может выключить, компьютер продолжает звучать. Через 2 сек программа снова оживает и выполняется оператор NoSound. Звук выключается. Таким образом, результатом выполнения этих трех операторов будет звук частотой 300 гц продолжительностью 2 сек.

Рассмотрим работу программы:

```
USES CRT;
BEGIN
   Sound(900);Delay(1000);Sound(200);Delay(3000);NoSound
END.
```

Начинается она со звука частотой 900 гц и продолжительностью 1 с, а затем оператор *Sound(200)* включает вместо звука в 900 гц звук частотой 200 гц, который длится 3 с.

В операторах Sound и Delay вместо чисел можно записывать целочисленные переменные величины и выражения. Вот программа, производящая серию постепенно повышающихся звуков:

```
USES CRT;
VAR hz: Integer;
BEGIN
   hz:=60;
   while hz<800 do begin
      Sound(hz);
      Delay(1000);
      hz:=hz+40
   end;
   NoSound
END.
```

Если вас интересуют музыкальные ноты, то вот вам операторы Sound, задающие все ноты третьей октавы:

Нота до	Sound(523)
Нота до диез	Sound(554)
Нота ре	Sound(587)
Нота ре диез	Sound(622)
Нота ми	Sound(659)
Нота фа	Sound(698)
Нота фа диез	Sound(740)
Нота соль	Sound(784)
Нота соль диез	Sound(831)
Нота ля	Sound(880)
Нота ля диез	Sound(932)
Нота си	Sound(988)

Задания 59-63:

59) Уменьшив как следует числа в операторах *Delay(1000)* и *hz:=hz+40* можно добиться впечатления одного непрерывного постепенно повышающегося звука (сирена). Попробуйте сделать это.

60) Если вам это удалось, попробуйте смоделировать сирену милицейской машины: звук вверх - звук вниз - звук вверх - звук вниз - … и так несколько раз.

61) Быстро чередуя короткие звуки двух разных частот и короткие паузы, можно добиться разных звуковых эффектов и шумов, например звука телефонного звонка или моторчика авиамодели.

62) Сделайте “датчик чувствительности уха к высоким частотам”. Известно, что человеческое ухо не может слышать звуки, частота которых превышает 10000-20000 гц. У разных людей порог чувствительности разный. Напишите программу, которая выдает звуки все более высокой частоты и печатает на экране значения этой частоты, так что человек успевает увидеть, при какой частоте он перестает слышать звук. Вы сможете определить, у кого порог чувствительности выше – у вас или у вашего друга. Удобно внутри цикла использовать команду ReadLn.

63) Если у вас есть некоторое музыкальное образование, вы можете попробовать заставить компьютер исполнить простенькую мелодию из нескольких нот.

8.2. Простейшие процедуры

Задача: Все вы слышали про азбуку Морзе, широко использовавшуюся раньше для радиосвязи с кораблями и не только с ними. Включив радиоприемник и покрутив ручку настройки, можно было услышать частую тоненькую дробь однотонных сигналов разной длительности: *точки* (очень короткие сигналы) и *тире* (сигналы подлиннее). Каждая буква алфавита кодируется в азбуке Морзе последовательностью точек и тире.

Вот таблица кодирования русских и латинских букв (латинские буквы показаны строчными, а русские - заглавными):

Aa	.-	Бb	-…	Bw	.--	Гg	--.	Дd	-..	Ee	.	Жv	…-	Зz	--..
Ии	..	Йj	.---	Kk	-.-	Лl	.-..	Mm	--	Hn	-.	Oo	---	Пр	.--.
Pr	.-.	Cs	…	Tt	-	Уu	..-	Фf	..-.	Xh	….	Цc	-.-.	Ч	---.
Ш	----	Щq	--.-	Ъ,Ьx	-..-	Ыy	-.--	Э	..-..	Ю	..--	Я	.-.-		

Закодируем азбукой Морзе текст *PIPING* и заставим компьютер воспроизвести соответствующий набор звуков. Пусть продолжительность точки - 100 миллисекунд, тире - 200, пауза молчания после точки или тире - 80, пауза после буквы - 300. Частота звуков не играет роли, выберем наугад 900 гц.

Вот как будет выглядеть фрагмент, воспроизводящий точку:

```
Sound(900);Delay(100); NoSound; Delay(80)
```

А вот фрагмент, воспроизводящий тире:

```
Sound(900);Delay(200); NoSound; Delay(80)
```

Вот вся программа:

Программа 1

```
USES CRT;
BEGIN
            {буква P:}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Sound(900); Delay(200); NoSound; Delay(80);  {тире}
  Sound(900); Delay(200); NoSound; Delay(80);  {тире}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Delay(300);                                  {пауза}
            {буква I:}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Delay(300);                                  {пауза}
            {буква P:}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Sound(900); Delay(200); NoSound; Delay(80);  {тире}
  Sound(900); Delay(200); NoSound; Delay(80);  {тире}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Delay(300);                                  {пауза}
            {буква I:}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Delay(300);                                  {пауза}
            {буква N:}
  Sound(900); Delay(200); NoSound; Delay(80);  {тире}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Delay(300);                                  {пауза}
            {буква G:}
  Sound(900); Delay(200); NoSound; Delay(80);  {тире}
  Sound(900); Delay(200); NoSound; Delay(80);  {тире}
  Sound(900); Delay(100); NoSound; Delay(80);  {точка}
  Delay(300);                                  {пауза}
END.
```

Недостатки программы:

- Довольно большой объем, что обидно, так как в программе много одинаковых фрагментов.
- Если бы не комментарии, было бы совершенно непонятно, о чем эта программа.

А теперь я напишу ту же программу, но с использованием процедур:

Программа 2

```
USES CRT;

PROCEDURE  tochka;
   BEGIN Sound(900); Delay(100); NoSound; Delay(80) END;

PROCEDURE  tire;
   BEGIN Sound(900); Delay(200); NoSound; Delay(80) END;
```

```
BEGIN
   {буква P:}  tochka;  tire;  tire;  tochka;   Delay(300);
   {буква I:}  tochka;  tochka;                  Delay(300);
   {буква P:}  tochka;  tire;  tire;  tochka;   Delay(300);
   {буква I:}  tochka;  tochka;                  Delay(300);
   {буква N:}  tire;  tochka;                    Delay(300);
   {буква G:}  tire;  tire;  tochka;             Delay(300);
END.
```

Программа 2 гораздо короче и даже без комментариев понятнее программы 1. Поясним, как мы получили ее из предыдущей.

Сначала мы обнаружили в программе 1 часто повторяющиеся фрагменты. Их было два:

Sound(900); Delay(100); NoSound; Delay(80); {точка}

Sound(900); Delay(200); NoSound; Delay(80); {тире}

Затем мы придумали имена каждому фрагменту: *tochka* и *tire*. После этого можно было писать программу 2. Каждый фрагмент мы записали один раз в начале программы выше главного BEGIN, оформив его в виде так называемого **описания процедуры**:

```
PROCEDURE  tochka;
  BEGIN Sound(900); Delay(100); NoSound; Delay(80) END;
PROCEDURE  tire;
  BEGIN Sound(900); Delay(200); NoSound; Delay(80) END;
```

В результате программа "узнала", что такое *tochka* и *tire*. С этого момента **имена процедур** *tochka* и *tire* можно употреблять, как обыкновенные операторы, ниже главного BEGIN. Паскаль выполняет программу, начиная с главного BEGIN, и когда он натыкается на имя процедуры, он подставляет вместо него соответствующий фрагмент, взятый из описания процедуры. Это событие называется **вызовом** процедуры или **обращением** к процедуре.

Синтаксис описания простейшей процедуры таков:

PROCEDURE имя **;** **BEGIN** оператор **;** оператор **;** ... **END**

Слово PROCEDURE читается "про'сидже", переводится "процедура". Имя процедуры создается по тем же правилам, что и имя переменной. Все, что идет после имени, будем называть телом процедуры.

Задание 64: Составьте программу с процедурами, которая исполнит мелодию "Чижик-пыжик" (ми-до-ми-до-фа-ми-ре-соль-соль-ля-си-до-до-до).

А теперь попробуем еще больше упростить нашу программу. Замечаем, что и в программе 2 тоже имеются одинаковые фрагменты:

{буква P:} *tochka; tire; tire; tochka;   Delay(300);*

{буква I:} *tochka; tochka;   Delay(300);*

Для экономии места их тоже выгодно оформить в виде процедур:

```
PROCEDURE  P;
  BEGIN tochka;  tire;  tire;  tochka;   Delay(300) END;
PROCEDURE  I;
  BEGIN tochka;  tochka;                 Delay(300) END;
```

Остальные буквы тоже выгодно оформить в виде процедур, но уже не для экономии места, а для удобочитаемости программы. Вот окончательный вариант программы:

Программа 3

```
USES CRT;

PROCEDURE  tochka;
   BEGIN Sound(900); Delay(100); NoSound; Delay(80) END;

PROCEDURE  tire;
   BEGIN Sound(900); Delay(200); NoSound; Delay(80) END;

PROCEDURE  P;
   BEGIN tochka;  tire;  tire;  tochka;   Delay(300) END;

PROCEDURE  I;
   BEGIN tochka;  tochka;                 Delay(300) END;
```

```
PROCEDURE  N;
   BEGIN tire;  tochka;              Delay(300) END;

PROCEDURE  G;
   BEGIN tire;  tire;  tochka;       Delay(300) END;


BEGIN
   P;I;P;I;N;G
END.
```

Эта программа понятна и без комментариев. От предыдущей она отличается тем, что процедуры вызываются не только из тела программы, но и из тел других процедур. Действительно, посмотрим на самое начало выполнения программы. Первое, на что натыкается Паскаль ниже главного BEGIN, это *P*. Заглянув выше главного BEGIN, Паскаль обнаруживает, что *P* - это имя процедуры, и начинает ее выполнять (вызывает на выполнение). При этом, первое, на что он натыкается, это *tochka*. Заглянув выше, Паскаль обнаруживает, что *tochka* - это имя процедуры, и тоже начинает ее выполнять (вызывает на выполнение). Обратите внимание, что

вызываемая процедура должна быть описана выше вызывающей.

Такая уж особенность у Паскаля. О том, что делать, если это требование выполнить невозможно, написано в 10.6. Закончив выполнять процедуру *tochka*, Паскаль возвращается в процедуру *P* (возвращает управление процедуре *P*). Там он идет к следующему оператору. Это оказывается *tire*. И т.д.

Задание 65: Компьютер печатает текст ***Песня «Чижик-пыжик». 1 куплет***. После этого исполняется мелодия чижика-пыжика (см. выше). Затем компьютер печатает текст ***2 куплет*** и мелодия чижика-пыжика исполняется еще раз.

8.3. Процедуры и операторы

Разберемся немного с терминологией. Прежде всего уточним структуру программы на Паскале. Любая программа на Паскале состоит из двух разделов:

```
   {Раздел описаний}
BEGIN
   {Раздел операторов}
END.
```

Раздел описаний может и отсутствовать, если же он присутствует, то может содержать разделы VAR, LABEL, USES, PROCEDURE и другие, нами еще не изученные.

Раздел операторов состоит из операторов, разделенных точками с запятой.

В самом начале части II я упомянул, что многие операторы на Паскале являются обращениями к процедурам. Для простоты мы договорились не различать поначалу обращения к процедурам и другие операторы. Сейчас нам пора этому научиться. Кроме процедур, написанных нами (таких как tochka, tire и др.), существуют еще так называемые **стандартные процедуры** Паскаля. Мы уже изучили следующие стандартные процедуры: Write, WriteLn, Read, ReadLn, Sound, Delay, NoSound. Они называются стандартными, потому что определены “внутри Паскаля”[7] и ими можно пользоваться, не описывая их, как мы описывали созданные нами “пользовательские” процедуры. Природа пользовательских и стандартных процедур едина, поэтому мы будем называть и те и другие просто процедурами. Операторами же мы будем продолжать называть операторы goto, if, case, for, while, repeat, оператор присваивания, составной оператор и еще незнакомые нам оператор with и пустой оператор.

8.4. Стандартные процедуры Halt и Exit

Halt читается “хальт”, переводится с немецкого “стой”.

Exit читается “ ‘экзит”, переводится с английского “выход”.

До сих пор мы составляли программы, которые заканчивали свою работу на END с точкой, не раньше. Процедура **Halt** заставляет Паскаль завершить работу программы, не доходя до конечного END с точкой. Пример: программа

[7] точнее, внутри стандартных модулей Паскаля, о которых вы узнаете позже

```
BEGIN    Write(1);  Write(2);  Halt;  Write(3)  END.
```

напечатает 12, а программа

```
PROCEDURE a;  BEGIN  Write(6);  Halt;  Write(7); END;
BEGIN  Write(2);  a;  Write(3);  Halt;  Write(4)  END.
```

напечатает 26.

Обращение к процедура **Exit**, если оно встречается в процедуре, заставляет Паскаль вернуться в процедуру, ее вызвавшую. Пример: заменим в предыдущей программе первый из двух Halt на Exit и посмотрим, что будет. А будет то, что программа

```
PROCEDURE a;  BEGIN  Write(6);  Exit;  Write(7); END;
BEGIN  Write(2);  a;  Write(3);  Halt;  Write(4)  END.
```

напечатает 263.

Если Exit встречается в разделе операторов программы, то он, подобно Halt, вызывает выход из программы.

Задание 66:

Вот вам программа с процедурами. Вам нужно, не запуская ее, записать на бумажке весь разговор, который ведут герои “Трех мушкетеров”.

```
PROCEDURE ATOS;
   BEGIN   WriteLn ('Я - Атос')                                          END;
PROCEDURE ARAMIS;
   BEGIN   WriteLn ('Это так же верно,как то,что я -Арамис!')            END;
PROCEDURE PORTOS;
   BEGIN   WriteLn ('А я Портос! Я правильно говорю, Арамис?');
           Aramis;
           WriteLn ('Он не врет, ваше величество! Я Портос, а он Арамис.') END;
PROCEDURE DARTANIAN;
   BEGIN   WriteLn ('А я все думаю, ваше величество - куда девались подвески королевы?');
           Exit;
           WriteLn ('Интересно, что ответит король?');
           Portos                                                        END;
BEGIN
   WriteLn ('Я, король Франции, спрашиваю вас - кто вы такие? Вот ты - кто такой?');
   Atos;
   WriteLn ('А ты, толстяк, кто такой?');
   Portos;
   WriteLn ('А ты что отмалчиваешься, усатый?');
   dArtanian;
   WriteLn ('Анна! Иди-ка сюда!!!');
   Halt;
   WriteLn ('Аудиенция закончена, прощайте!');
END.
```

Выполнив задание, скопируйте программу в редактор Паскаля и запустите ее. Если ответ не сходится, запустите ее в отладочном пошаговом режиме.

Теперь вы достаточно знаете о процедурах, чтобы они стали для вас удобными кирпичиками для постройки программ. Более мощным средством являются процедуры с параметрами, о которых вы узнаете в Глава 13.

Глава 9. Графика

Если в предыдущих главах на экране изображались только буквы, цифры и прочие символы, то эта глава научит вас рисовать.

9.1. Стандартные модули

В 2.5 я говорил, что при работе с языком программирования мы реально работаем с комплексом программ, позволяющим программисту создавать собственные программы, пользуясь при этом значительным количеством готовых процедур (а сейчас мы уточним - стандартных процедур[8]). На данный момент мы знаем следующие стандартные процедуры языка Паскаль: Write, WriteLn, Read, ReadLn, Sound, Delay, NoSound, Exit, Halt. Вспомним теперь, что для того, чтобы вообще быть выполненной, программа должна находиться в оперативной памяти компьютера. Паскаль – это программа. Значит, для того, чтобы мы могли работать с Паскалем, все стандартные процедуры Паскаля тоже должны находиться в памяти? Однако, память - вещь дефицитная и ее нужно экономить. Хорошо бы в память загружались не все процедуры, а только те, которые нужны данному программисту. Действительно, во многих приведенных в этой книге программах мы не использовали звук, зачем же нам было иметь в памяти процедуры Sound и NoSound?

Реально так и сделано. В Паскале определен целый ряд необязательных его кусков, которые называются **стандартными модулями**. Упрощенно говоря, стандартный модуль является сборником процедур и других элементов, которые загружаются с диска в память только тогда, когда программист специально попросит. Один из этих модулей называется **CRT** и кроме всего прочего занимается звуком. Другой называется **Graph** и дает возможность программисту работать с изображениями на экране. С некоторыми другими стандартными модулями мы познакомимся позже.

Если программист специально не попросил, то в памяти оказывается только минимальная часть Паскаля, которая нужна более-менее всем.

А как же попросить? Очень просто. Если вам нужно работать со звуком, вы пишете первой строкой своей программы - USES CRT, а если вы собираетесь работать с изображениями, то - USES Graph. USES читается “’юзез”, переводится “использует”. Значит, строка USES CRT является приказом компьютеру “использовать стандартный модуль CRT”, для чего загрузить в память те его процедуры и другие элементы, которые требует программа.

Вот так, довольно грубо я обрисовал идею использования стандартных модулей. Если же программист видит, что их возможностей ему не хватает, он может написать несколько процедур и создать из них свой собственный модуль, придумать ему имя и пользоваться затем точно так же, как стандартными. Об этом написано в 15.3.

9.2. Стандартный модуль Graph, текстовый и графический режимы

Graph читается “граф”, это сокращение слова “графика”. Если мы напишем первой строкой своей программы USES Graph, то Паскаль предоставит в наше распоряжение целый ряд процедур и других средств, позволяющих нам рисовать на экране разноцветные точки, отрезки прямых, дуги, закрашенные и незакрашенные окружности, прямоугольники, а также выполнять ряд других действий. Пользуясь этими возможностями, мы очень скоро напишем программы, рисующие причудливые картинки и заставляющие изображения двигаться по экрану. С помощью модуля CRT мы научимся управлять этим движением с клавиатуры, а значит сможем создавать свои собственные компьютерные игры. В 10.9 я объясню создание игры “Торпедная атака”.

А в этой главе я опишу работу самых популярных с моей точки зрения процедур модуля Graph.

9.2.1. Текстовый и графический режимы

Существуют два режима (способа) работы компьютера с монитором - текстовый и графический. В любом месте программы вы можете приказать компьютеру переключиться из одного режима в другой.

Текстовый режим используется для вывода на экран текстовой и числовой информации. Работая в текстовом режиме, компьютер считает экран разбитым на 25 строк и 80 столбцов, иногда на другое количество. В каждой из получившихся клеточек умещается ровно одна буква или цифра или знак препинания или любой другой символ. Какой именно символ будет находиться в клеточке, диктуете вы в вашей программе. Рисовать и показывать картинки компьютер в текстовом режиме не может.

Работая в **графическом режиме** компьютер считает экран разбитым на множество мельчайших пикселов, каждый из них - гораздо мельче клеточки текстового режима. Принцип построения изображения из пикселов описан в 3.4. В этом режиме компьютер может и рисовать картинки и печатать символы, причем если в текстовом

[8] и функций, и других элементов

режиме величина и форма всех букв более-менее одинакова, то в графическом режиме мы можем печатать символы самой разной формы и размеров.

Получается, что графический режим лучше текстового? В общем, да, конечно. Но у графического режима есть один недостаток - он требует от компьютера значительных усилий и поэтому на маломощных компьютерах часто работает раздражающе медленно.

Во всех написанных ранее программах мы ничего не говорили об этих режимах, мы молчали о них. Если мы в программе специально не просим, то Паскаль всегда выбирает текстовый режим. Говорят, что текстовый режим в Паскале используется **по умолчанию**.

9.2.2. Переключение между текстовым и графическим режимами

Итак, если вы собираетесь работать с изображениями, то должны переключиться в графический режим и для этого пишете первой строкой программы USES Graph. Но сама по себе эта строка не является указанием компьютеру переключиться в графический режим, хотя бы потому, что находится в разделе описаний, а не в разделе операторов. Для переключения в графический режим (или, как говорят, для **инициализации** графического режима) служит стандартная процедура InitGraph. Для того, чтобы закрыть графический режим и снова переключиться в текстовый, служит стандартная процедура CloseGraph.

(Начиная с этого момента я не буду приводить переводы и произношение английских слов. Многие из них приведены в приложении П3 и П4.)

Вот пример программы, которая сначала в текстовом режиме пишет на экране текст "*Это текстовый режим*", затем переключается в графический режим, рисует окружность, а затем снова переключается в текстовый режим и пишет "*Это снова текстовый режим*":

```
USES    Graph;
VAR     Device, Mode   : Integer;
BEGIN
   WriteLn('Это текстовый режим');
            {Инициализируем графический режим:}
   ReadLn;
   Device:=0;
   InitGraph(Device,  Mode,  '<путь к графическим драйверам>');
   Circle(100,80,50);  {Обращение к процедуре рисования окружности. Пока без пояснений}
   ReadLn;
            {Закрываем графический режим, что рекомендую:}
   CloseGraph;
   WriteLn('Это снова текстовый режим');
   ReadLn
END.
```

Пояснения: Перед использованием процедуры *InitGraph* необходимо создать две переменные величины типа Integer с произвольными именами (я использовал имена *Device* и *Mode*). Обе эти переменные при обращении к процедуре *InitGraph* должны быть записаны внутри круглых скобок. Вам на первых порах совершенно не обязательно это знать, но поясню, что *Device* означает тип вашего видеоадаптера (*CGA, EGA, VGA* или другой), а *Mode* означает номер графического режима. Если вы ничего не знаете ни о том, ни о другом, смело пишите *Device := 0* и Паскаль сам определит тип вашего видеоадаптера и установит самый мощный из допустимых графический режим.

Третий элемент - не что иное, как путь к графическим драйверам Паскаля. Я использовал угловые скобки <>, чтобы подчеркнуть, что в вашей программе нужно писать не те четыре русских слова, что внутри угловых скобок, а то, на что они указывают. Почти наверняка для вас графический драйвер представлен файлом *egavga.bgi*, расположенном в каталоге *BGI*. Если сам Паскаль расположен в каталоге *TP* диска *c*, то строка вашей программы будет выглядеть так:

InitGraph(Device, Mode, 'c:\TP\BGI')

Если вы ее записали верно, то можете попытаться запустить программу, и при правильной настройке Паскаля у вас все получится. Тех, кто не знает, что такое каталоги и файлы, отсылаю к приложению.

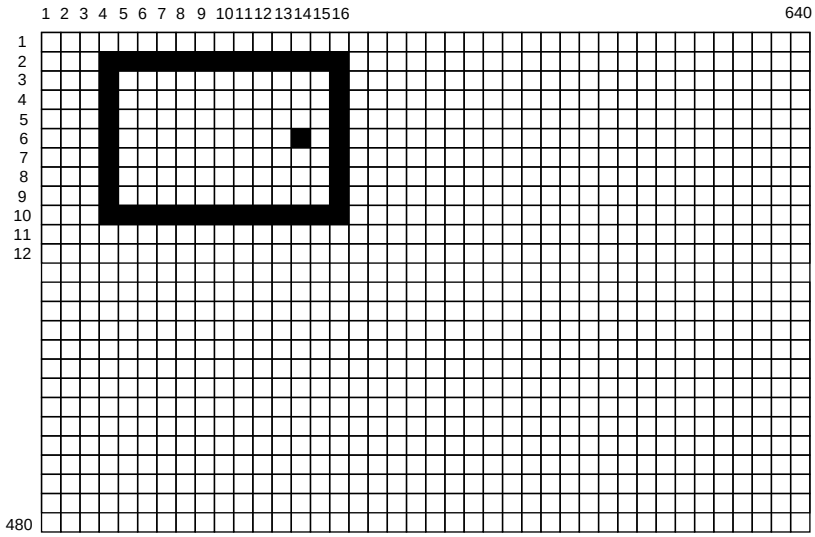
Если Паскаль при запуске графического режима жалуется (*File not found (GRAPH.TPU)*), то прочтите «Обзор популярных команд меню» из части IV. Там сказано, что делать.

Учтите, что при переключении режимов все содержимое экрана стирается.

9.3. Рисуем простейшие фигуры

Начиная с этого момента я буду для определенности считать, что графический режим вашего монитора считает экран разделенным на 640 пикселов в ширину и 480 в высоту и исходя из этого буду писать программы. Если у вас режим другой, то вы легко сможете сделать поправки в программах.

Попробуем нарисовать точку и прямоугольник. Пусть мы хотим их видеть в следующем месте экрана:



Для этого мы пишем обращения к двум процедурам:

Для **точки** - **PutPixel**(14,6, Yellow)

Для **прямоугольника** - **Rectangle**(4,2,16,10)

Как видим, при обращении к процедуре PutPixel в скобках нужно указывать три элемента, у Rectangle - четыре. Будем называть элементы, разделенные запятыми внутри скобок, **параметрами** процедур.

Смысл параметров процедур PutPixel и Rectangle ясен из рисунка. В PutPixel первый параметр - горизонтальная координата точки, второй - вертикальная, третий - цвет точки (желтый). В Rectangle первая пара параметров - координаты любого из углов прямоугольника, вторая пара - координаты противоположного ему угла (только не соседнего). В каждой паре первой идет горизонтальная координата, второй - вертикальная. Цвет прямоугольника задается в другой процедуре, о которой разговор позже.

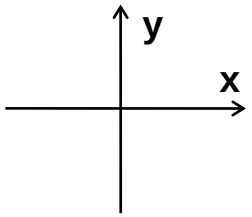
Модуль Graph позволяет удобно использовать всего лишь 16 цветов. Вот они:

0	Black	*Черный*
1	Blue	*Синий*
2	Green	*Зеленый*
3	Cyan	*Голубой*
4	Red	*Красный*
5	Magenta	*Фиолетовый*
6	Brown	*Коричневый*
7	LightGray	*Светлосерый*
8	DarkGray	*Темносерый*
9	LightBlue	*Ярко-синий*
10	LightGreen	*Ярко-зеленый*
11	LightCyan	*Ярко-голубой*
12	LightRed	*Розовый*
13	LightMagenta	*Малиновый*
14	Yellow	*Желтый*
15	White	*Белый*

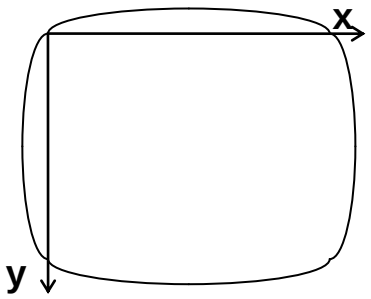
Стандартный русский перевод цвета в этой таблице не всегда точен. Например, LightMagenta только с большой натяжкой можно назвать малиновым цветом.

В обращениях к процедурам, имеющим дело с цветом, вместо английского названия цвета можно писать соответствующее число. Например, вместо *PutPixel(14,6, Yellow)* можно написать *PutPixel(14,6, 14)*.

В школе вы привыкли к такой системе координат:

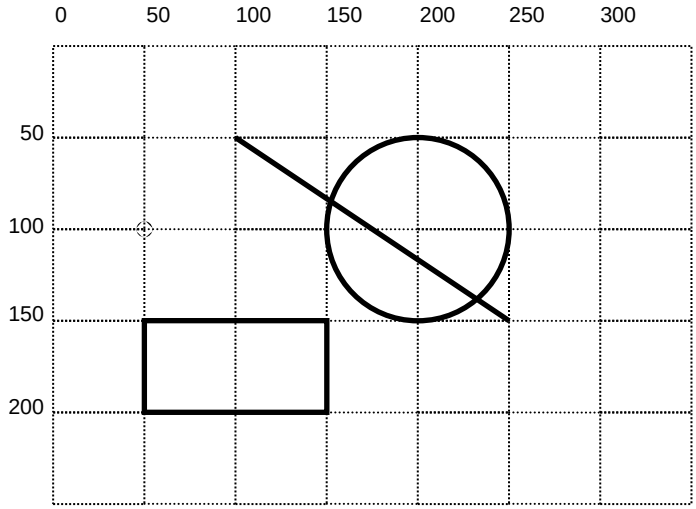


В компьютере же применяется такая:



Как видите, ось y направлена вниз. Это не очень привычно. Если вас это раздражает, то в 13.2 вам будет предложено исправить ситуацию.

Напишем программу, которая рисует точку, прямоугольник, окружность и отрезок прямой так, как это показано на рисунке:

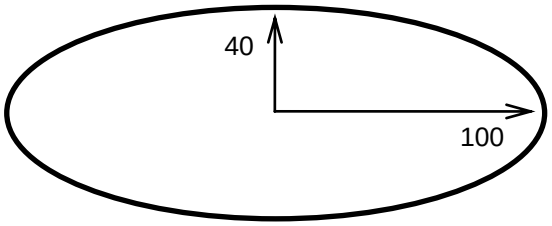



```
USES    Graph;
VAR     Device, Mode: Integer;
BEGIN
   Device:=0;
   InitGraph(Device,  Mode,  '<путь к графическим драйверам>');
   PutPixel(50,100,White);
   Rectangle(150,150,50,200); {правый верхний и левый нижний углы}
   Circle(200,100,50);        {окружность}
   Line(100,50,250,150);      {отрезок прямой}
   ReadLn;
   CloseGraph
END.
```

Пояснения: **Отрезок прямой** рисуется процедурой **Line**. Мы знаем, что отрезок прямой можно построить, если известно положение его двух крайних точек. Они-то и задаются в обращении к процедуре. Первая пара параметров - координаты одной точки (любой из двух), вторая пара - другой.

Окружность можно построить, если известно положение центра и радиус. Окружность рисуется процедурой **Circle**, первые два параметра которой - координаты центра, третий - радиус.

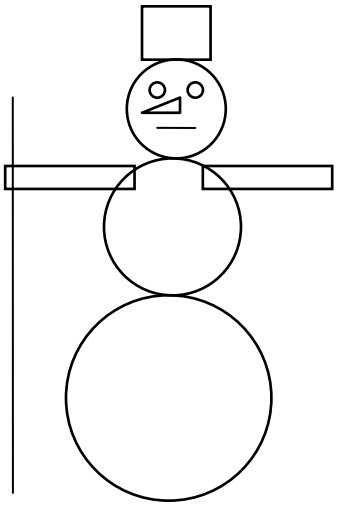
Попробуем теперь нарисовать **эллипс** с центром в точке $x=200$, $y=150$, вот такой:



Это выполнит процедура **Ellipse**. Вот ее вызов - *Ellipse(200,150,0,360,100,40)*.Если вы хотите нарисовать не полный эллипс, а только часть его, вместо 0 укажите начальный угол дуги эллипса, скажем 90, а вместо 360 - конечный угол, скажем 180.

Процедура **ClearDevice** стирает все с экрана в графическом режиме.

Задание 67: Нарисуйте снеговика:



9.4. Работа с цветом. Заливка. Стиль линий и заливки

Сначала поговорим о стиле линий. Если вы хотите, чтобы линии, которыми чертятся фигуры, были потолще, используйте процедуру **SetLineStyle**. Вот ее вызов - *SetLineStyle(0, 0, ThickWidth)*. Это приказ отныне рисовать линии толстыми. Если вы хотите вернуться к тонким линиям, напишите *SetLineStyle(0, 0, NormWidth)*.

Первый параметр процедуры SetLineStyle управляет стилем прямых отрезков (штриховые, пунктирные и др.). Попробуйте вместо 0 написать 1, 2 или 3 и посмотрите, что получится.

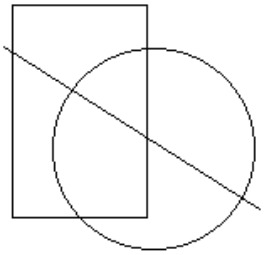
Второго параметра касаться не будем.

А теперь поговорим о цвете. Если цвет точки задается непосредственно в вызове процедуры, которая рисует точку (PutPixel), то цвета многих других фигур задаются по-иному. Делает это специальная процедура **SetColor**. Пример: *SetColor(Yellow)* приказывает компьютеру отныне рисовать фигуры желтым цветом.

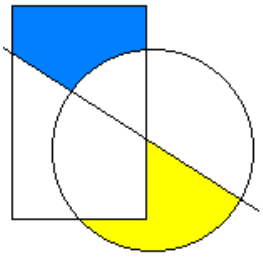
Вот поясняющий фрагмент:

```
Circle(100,100,20);     {окружность белая и тонкая, так как по умолчанию
                         {Паскаль все рисует белым цветом тонкими линиями}
SetColor(Yellow);
Circle(150,100,20);     {желтая тонкая окружность }
SetLineStyle(0, 0, ThickWidth);
Circle(200,100,20);     {желтая толстая окружность}
SetColor(Blue);
SetLineStyle(0, 0, NormWidth);
Circle(250,100,20);     {синяя тонкая окружность}
```

Работая в Паскале, вы можете покрасить любым цветом (залить краской) любую область экрана внутри замкнутого контура. Пусть вы нарисовали такую картинку:



После этого вы можете ее покрасить, например, вот так:



Как красить? Начнем с процедуры **SetFillStyle**. Подобно тому, как процедура SetColor сама ничего не рисует, а только устанавливает цвет линий на будущее, процедура SetFillStyle сама ничего не заливает, а устанавливает цвет и стиль заливки на будущее. Например, *SetFillStyle(1, Green)* устанавливает заливку зеленым цветом. На единичку пока не обращаем внимания.

Сам процесс заливки вызывает процедура **FloodFill**. Например, *FloodFill(200, 150, Blue)* вызывает следующие действия Паскаля:

В точке экрана с координатами $x=200$ и $y=150$ встает “маляр с ведром краски”, цвет которой был указан в обращении к процедуре SetFillStyle (в нашем примере - зеленый), и начинает эту краску разливать вокруг себя. Краска свободно растекается по экрану и преградой для нее становится только линия, имеющая цвет, указанный в обращении к процедуре FloodFill, то есть синий. Очевидно, если синий контур не будет замкнут вокруг “маляра”, то краска прольется на весь экран.

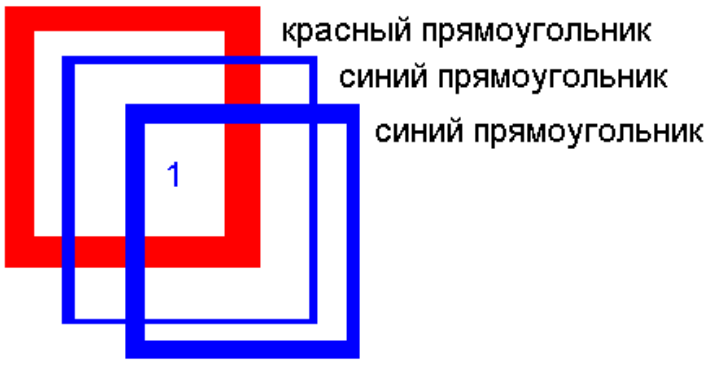
Пример: Следующий фрагмент нарисует желтый квадрат и зальет его красным цветом:

```
SetColor(Yellow);
SetFillStyle(1, Red);
Rectangle(200,50,300,150);
FloodFill(250, 100, Yellow)
```

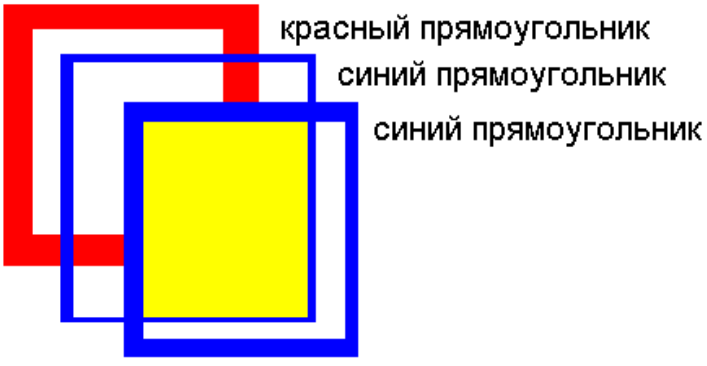
А теперь попробуем поставить “маляра” вне квадрата, записав в нашем фрагменте вместо *FloodFill(250, 100, Yellow)* например, *FloodFill(250, 200, Yellow)*. Теперь красной будет вся поверхность экрана, кроме области внутри квадрата.

Распространенная ошибка новичков - на вопрос о том, каким цветом будет красить оператор *FloodFill(250, 100, Yellow)*, они отвечают “Желтым”, хотя нужно отвечать “Не знаю, надо посмотреть в SetFillStyle”.

Напомним, что в нашем случае только желтые линии и области являются преградой для красной краски. Например, попытаемся покрасить желтой краской маленький прямоугольник, помеченный единичкой, полученный пересечением больших красного и синего прямоугольников:



Для этого поместим “маляра” внутрь этого прямоугольника и используем операторы *SetFillStyle(1, Yellow)* и *FloodFill(… , … ,Blue)*. Мы получим следующую картину:



Очевидно, при помощи FloodFill нужный прямоугольник закрасить невозможно, так как он ограничен линиями разного цвета.

Заливку можно делать не только сплошной, но и по-разному заштрихованной. Для этого вместо единички в обращении к процедуре SetFillStyle попробуйте другие целые числа - 2, 3 и др.

Задание 68: Усовершенствуйте снеговика из 9.3. Он должен стать таким - с красным носом, красными губами, желтой шапкой и толстым синим посохом. При помощи оператора WriteLn сделайте надпись “Это снеговик”.

9.5. Используем в рисовании переменные величины

Если вы нарисовали снеговика, то наверное согласитесь, что для этого вам пришлось основательно потрудиться, хотя сам рисунок не слишком богат, в нем всего порядка десяти элементов.

Как заставить Паскаль короткой программой рисовать множество элементов? Ответ: применять циклы, используя в обращениях к графическим процедурам вместо чисел переменные величины и арифметические выражения.

Задача: Нарисовать горизонтальный ряд окружностей радиусом 10 на расстоянии 100 от верхнего края экрана и с такими горизонтальными координатами 50, 80, 110, 140, **...** , 290.

Как видим, центры соседних окружностей отстоят друг от друга на 30. Вот примитивный фрагмент, решающий эту задачу:

```
Circle( 50,100,10);
Circle( 80,100,10);
Circle(110,100,10);
Circle(140,100,10);
Circle(170,100,10);
Circle(200,100,10);
Circle(230,100,10);
Circle(260,100,10);
Circle(290,100,10);
```

При вводе этой программы вас будет раздражать необходимость вводить много раз почти одно и то же. Воспользуйтесь копированием, которое объяснено в параграфе «Копирование и перемещение фрагментов текста» из части IV.

Мы видим, что здесь Паскаль 9 раз выполнит одну и ту же процедуру, причем при каждом следующем обращении первый параметр вырастает на 30.

А теперь решим эту же задачу при помощи цикла.

Придумаем для первого параметра переменную величину, например, x. Чтобы x изменялся, организуем цикл repeat. Вот программа решения задачи:

```
USES    Graph;
VAR     x, Device, Mode :Integer;
BEGIN
   Device:=0;
   InitGraph(Device,  Mode,  '<путь к графическим драйверам>');
   x:=50;
   repeat
      Circle(x,100,10);
      x:=x+30;
   until x>290;
   ReadLn;
   CloseGraph
END.
```

Задание 69:. Попробуйте уменьшить расстояние между центрами окружностей, не изменяя их радиуса, нарисовав их плотнее, чтобы они пересекались, еще плотнее, пока они не образуют “трубу”.

Задание 70:. Удлините трубу налево и направо до краев экрана.

Задание 71:. Увеличьте толщину трубы.

Заставим окружности вести себя посложнее. Например, расположим их не по горизонтали, а по диагонали экрана в направлении от левого верхнего угла в правый нижний. Для этого организуем еще одну переменную - вертикальную координату y - и заставим ее тоже изменяться одновременно с x.

```
USES    Graph;
VAR     x, y, Device, Mode : Integer;
BEGIN
   Device:=0;
   InitGraph(Device,  Mode,  '<путь к графическим драйверам>');
   x:=50;
   y:=20;
   repeat
      Circle(x,y,10);
      x:=x+30;
      y:=y+20;
   until x>290;
   ReadLn;
   CloseGraph
END.
```

Если мы захотим менять радиус, то организуем переменную R, тоже типа Integer.

Задание 72:. Нарисуйте ряд точек по направлению из левого нижнего угла в правый верхний.

Задание 73:. *"Круги на воде"*. Нарисуйте пару десятков концентрических окружностей, то есть окружностей разного радиуса, но имеющих общий центр.

Задание 74:. *"Компакт-диск"*. Если радиус самого маленького "круга на воде" будет порядка 50, а самого большого - во весь экран, и если радиусы соседних окружностей будут различаться на 2-3 пиксела, то на экране вы увидете привлекательный "компакт-диск". Сделайте его золотым (Yellow).

Задание 75:. Не трогая x, а меняя только y и R, вы получите коническую башню.

Задание 76:. Меняя все три параметра, вы получите трубу, уходящую в бесконечность.

Задание 77:. Разлинуйте экран в линейку.

Задание 78:. А теперь в клетку.

Задание 79:. А теперь в косую линейку.

Задание 80:. Начертите ряд квадратов.

Чтобы получить богатые рисунки, нужно использовать богатые возможности Паскаля: вложенные циклы, ветвление внутри цикла и т.д., например:

Задание 81:. Нарисуйте шахматную доску.

Задание 82:. *"Ковер"*. В задании 69 вы рисовали горизонтальный ряд пересекающихся окружностей. Теперь нарисуйте один под другим много таких рядов.

Указания: Здесь вам понадобятся вложенные циклы. Если центры соседних окружностей отстоят друг от друга на одинаковое расстояние что по горизонтали, что по вертикали, и если удачно подобраны остальные числа, то у вас получится красивый ковер во весь экран с аккуратными краями.

Задание 83:. Пусть у этого ковра будет вырезан левый нижний угол.

Задание 84:. **...** и вдобавок вырезан квадрат посередине.

9.6. Использование случайных величин при рисовании

Как получить случайное число? Давайте сначала напечатаем его. Для этого подойдет функция Random из 4.9. WriteLn(Random(100)) напечатает целое неотрицательное число, какое - мы заранее не знаем, знаем только, что меньше 100. Легко догадаться, что *WriteLn (500 + Random(100))* напечатает случайное число из диапазона от 500 до 599.

Попробуем нарисовать "*звездное небо*". Для этого достаточно в случайных местах экрана нарисовать некоторое количество разноцветных точек (скажем, 1000). Точка ставится процедурой PutPixel. Как сделать координаты и цвет точки случайными? Тот же Random. Если ваш экран имеет размер 640×480 пикселов, то обращение к процедуре рисования одной точки случайного цвета будет выглядеть так:

PutPixel (Random(640), Random(480), Random(16))

Число 16 взято по той причине, что все цвета в Паскале имеют номера от 0 до 15.

Для того, чтобы таких точек было 1000, используем цикл for:

```
for i:=1 to 1000 do     PutPixel  (Random(640),   Random(480),   Random(16))
```

Имейте в виду, что сколько бы раз вы не запускали программу с указанным фрагментом, картина созвездий на экране будет абсолютно одинакова. Если вам нужно, чтобы от запуска к запуску набор значений случайной величины менялся (а значит и созвездия), употребите разик до использования функции Random процедуру **Randomize**. Вот так:

```
Randomize;
for i:=1 to 1000 do     PutPixel  (Random(640),   Random(480),   Random(16))
```

Задание 85:. *"Дождь в луже"*. Заполните экран окружностями радиуса 20 в случайных местах.

Задание 86:. *"Цирк"*. То же самое случайных радиусов и цветов.

Задание 87:. *"Звезды в окне"*. Звездное небо в пределах прямоугольника.

9.7. Движение картинок по экрану

Идею создания иллюзии движения картинок по экрану я объяснил в 2.8. Попробуем заставить двигаться по экрану слева направо окружность. Для этого мы должны сначала нарисовать слева окружность и тут же стереть ее, для чего нарисовать ее на том же месте, но черным цветом. Несмотря на то, что мы окружность тут же стерли, она успеет мелькнуть на экране, и глаз это заметит. Затем нужно нарисовать и стереть такую же окружность чуть правее, затем еще правее и т.д. Вот программа:

```
USES    Graph;
VAR     x, Device, Mode    : Integer;
BEGIN
   Device:=0;
   InitGraph(Device,   Mode,   '<путь к графическим драйверам>');
   ReadLn;          {Переключение в графический режим иногда занимает одну-две секунды, поэтому, если вы хотите увидеть движение с самого начала, щелкните по клавише ввода через пару секунд}
   x:=40;
   repeat
      SetColor(White);
      Circle(x,100,10);    {Рисуем белую окружность}
      SetColor(Black);
      Circle(x,100,10);    {Рисуем черную окружность}
      x:=x+1               {Перемещаемся немного направо}
   until x>600;
   CloseGraph
END.
```

Когда вы попробуете выполнить эту программу на компьютере, изображение движущейся окружности может получиться некачественным - окружность в процессе движения может мерцать и пульсировать. Это связано с разверткой электронно-лучевой трубки вашего монитора. Попробуйте изменить радиус окружности или шаг движения по горизонтали или введите между рисованием и стиранием окружности небольшую паузу процедурой *Delay* - ситуация почти наверняка улучшится.

Задание 88:. Измените скорость движения. Если окружность движется небыстро, увеличьте скорость (*x:=x+2*), если слишком быстро – уменьшите процедурой *Delay*.

Задание 89:. Пусть одновременно движутся две окружности.

Задание 90:. Одна вниз, другая направо.

Задание 91:. Заставьте окружность отскочить от правого края экрана.

Задание 92:. Заставьте окружность бесконечно двигаться, отскакивая от правого и левого краев экрана.

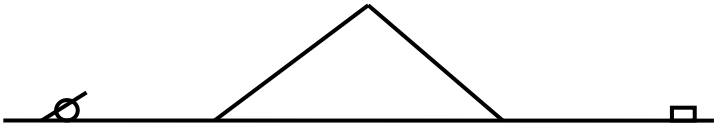
Если вы будете рисовать окружности толстой линией, то увидите, что движение сильно замедлилось, так как толстая линия рисуется гораздо дольше тонкой. То же относится и к закрашенной окружности.

Задание 93: *"Биллиардный шар".* Нарисуйте «биллиардный стол» – большой прямоугольник. Окружность под углом летает по столу, отскакивая от его краев по закону отражения. Попав "в лузу" (любой из четырех углов стола), останавливается.

Задание 94: Изобразите полет камня, брошенного с башни, для задания из 6.3. Напоминаю условие задания. Камень бросили горизонтально со 100-метровой башни со скоростью v=20м/с. Его расстояние от башни по горизонтали (s) выражается формулой s=v*t, где t – время полета камня в секундах. Высота над землей h выражается формулой $h=100 - 9.81*t^2/2$. Нарисуйте башню, землю, камень (маленькая окружность). Затем камень летит. Добейтесь, чтобы время полета камня на экране примерно соответствовало реальному времени, полученному в 6.3. Нарисуйте траекторию полета камня. Для этого достаточно, чтобы камень оставлял за собой следы в виде точек.

Указание: В задаче говорится о метрах, а на экране расстояние измеряется в пикселах. Поэтому вам придется задать масштаб, то есть вообразить, что один пиксел равен, скажем, одному метру. Тогда высота башни будет равна 100 пикселам, а скорость камня – 20 пикселов в секунду Правда, картинка на экране в этом случае может показаться вам маловатой. Тогда можете задать другой масштаб – один метр, скажем, - четыре пиксела. Тогда высота башни будет равна 400 пикселам, скорость камня – 80 пикселов в секунду, а формула изменится - $h=4*(100 - 9.81*t^2/2)$.

Задание 95 (сложное): Сделайте игру: Пушка на экране стреляет в цель ядрами. С какого выстрела она поразит противника? Между пушкой и целью расположена небольшая гора. Перед началом игры случайно задается горизонтальная координата цели. Затем рисуется картинка.



Перед каждым выстрелом компьютер отображает на экране номер выстрела и запрашивает у человека стартовую скорость ядра v и угол α наклона ствола пушки к земле. Затем летит ядро. Полет ядра подчиняется двум уравнениям: $s=v*t*\cos\alpha$ и $h=v*t*\sin\alpha - 9.81*t^2/2$ (см. предыдущее задание). Считается, что цель поражена, если ядро «отгрызло» от нее хоть маленький кусочек.

Я рассмотрел основные простые возможности модуля Graph. Некоторые другие средства модуля будут рассмотрены в Глава 15.

Глава 10. Создаем первую большую программу

В этой главе я вместе с вами напишу первую реальную программу. Достаточно большую для начинающих (строк на сто).

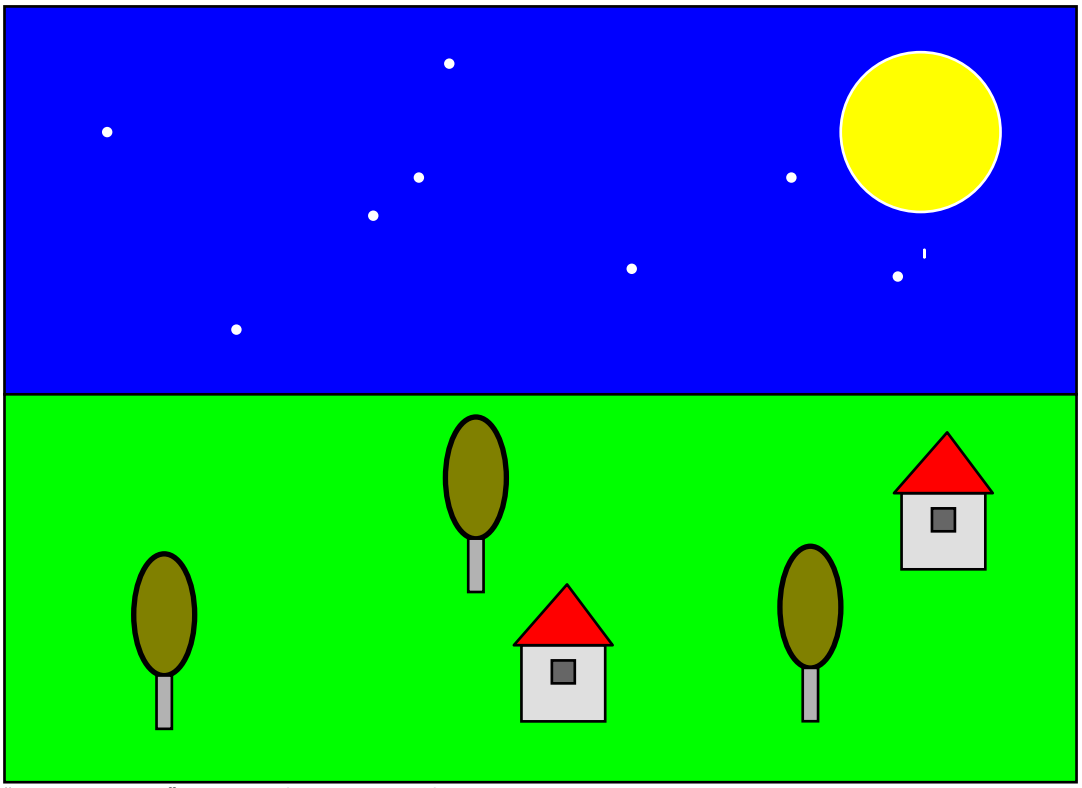
Прежде, чем приступить к большой программе, прочтите и изучите все, что вы еще не успели изучить в части IV, особенно "Отладку больших программ".

В 4.11 я уже разбирал порядок составления программы. Он подходит для небольших программ, не использующих процедуры. Однако, все реальные программы используют процедуры, поэтому этот порядок нужно дополнить.

10.1. Постановка задачи

Программу мы будем составлять для следующей задачи.
Задача: Создать мультфильм следующего содержания:

1) На экране возникает ночной пейзаж и секунды три ничего не происходит:



2) Затем звучит "космическая" музыка (секунд на 5).
3) Затем сверху слева появляется летающая тарелка (пусть это будет небольшой эллипс) и не спеша опускается на землю (до линии горизонта).
4) Затем снова раздается та же музыка.
5) Затем в окне ближайшего дома вспыхивает свет. Все.

10.2. Программирование по методу "сверху-вниз"

Повторю, что любая реальная (то есть достаточно большая и делающая «реальные» вещи) программа на Паскале содержит процедуры. Обычно более "основные" процедуры (P, I, N ...) обращаются к менее "основным"

(tochka, tire), те - к еще менее “основным” и т.д. Такая организация называется **иерархией**. Она подобна организации подчинения в армии - генералы приказывают полковникам, те - майорам и т.д.

Первая задача, стоящая перед программистом, - разбить программу на несколько частей, исходя из смысла задачи. В 8.2 это были части P, I, P, I, N, G. Обычно для правильного разбиения нужен опыт. В нашем случае разбиение напрашивается само собой исходя из последовательности действий мультфильма:

1) рисование пейзажа
2) пауза три секунды
3) музыка
4) движение тарелки
5) музыка
6) свет в окне

Теперь, когда мы разбили программу на части, нужно решить, какую из частей сделать процедурой, а какую можно описать просто группой операторов. Процедура выгодна тогда, когда часть или достаточно сложная (пейзаж), или встречается больше одного раза (музыка).

Итак, у нас будет три процедуры. Придумаем им имена:

* пейзаж - Landscape
* музыка - Music
* тарелка - Flying_Saucer

Теперь относительно каждой из процедур нужно решить, следует ли ее разбивать на части, причем рассуждать нужно так же, как при разбиении на части всей программы:

Music. Для музыки можно было бы использовать процедуры-ноты, но я для простоты ограничусь набором операторов Sound и Delay. Не разбиваем.

Flying_Saucer. Движение тарелки по экрану ничем не отличается от движения окружности, которое программируется небольшим циклом. Здесь тоже не нужно разбиения на части.

Landscape. Вот здесь без разбиения не обойтись. Для этого посмотрим, из чего состоит пейзаж. Он состоит из земли и неба. Земля состоит из горизонта, зеленой краски травы, домов и деревьев. Небо состоит из синей краски неба, звезд и луны. Вы можете разделить пейзаж на две большие части - землю и небо, каждую из которых делить дальше. Это будет логично и четко. Я же для простоты забуду о земле и небе и буду считать пейзаж прямо состоящим из горизонта, травы, домов, деревьев, синевы, звезд и луны. Порядок рисования я выберу такой:

1) горизонт
2) синева
3) луна
4) звезды
5) дерево
6) дерево
7) дерево
8) дом
9) дом
10) трава

Ни одна из этих десяти частей не является достаточно большой, чтобы претендовать на ранг процедуры. Однако, дом и дерево встречаются не один раз, значит они и будут процедурами. Придумаем им имена:

- дом - House
- дерево - Tree

Я на примере показал, как нужно дробить любую задачу на все более мелкие части до тех пор, пока не станет ясно, как любую из этих мелких частей запрограммировать на Паскале. Это и называется **программированием по методу “сверху вниз”**. Можно **программировать и по методу “снизу вверх”**, когда программист сначала составляет список самых мелких частей, из которых состоит задача, а потом собирает из них более крупные части.

Поговорим о том, как запрограммировать на Паскале наши самые мелкие части:

- горизонт - просто линия во весь экран
- синева - заливаем верхнюю часть экрана до горизонта синей краской
- луна - желтая окружность, залитая желтой краской
- звезды - звездное небо мы уже делали. Здесь надо будет только позаботиться, чтобы звезды не появлялись ниже горизонта
- дерево - залитые эллипс и прямоугольник
- дом - аналогично дереву
- трава - аналогично синеве

У вас может возникнуть следующий вопрос: Процедура Tree рисует некое дерево в определенном месте экрана. Как, использовав эту процедуру три раза, мы получим три дерева в разных местах экрана? Ответ: здесь нам на помощь придут переменные величины, о чем позже.

10.3. Сначала – работа над структурой программы

Прежде чем заселять жителей в город, нужно на каждом доме написать, кто в нем живет, и вообще, проследить, чтобы трамваи ходили правильно. А то возникнет неразбериха и люди не смогут ездить друг к другу в гости.

Прежде чем наполнять паскалевским содержимым наши части и процедуры, нужно сделать “скелет” программы, то есть “пустую” программу, в которой все части и процедуры ничего не делают, а только рапортуют о своем существовании, примерно так: “Работает процедура Дом”. Когда мы запустим такую программу-скелет и увидим, что все части и процедуры рапортуют в нужном порядке, мы можем надеяться, что ничего не перепутали и структура программы правильная. Вот после этого и можно постепенно заселять жителей, то есть наполнять пустые процедуры реальным содержанием. Если мы не проделаем этого заранее, то при отладке не будем знать, отчего у нас на экране получается ерунда - то ли оттого, что мы неправильно запрограммировали процедуру, то ли от того, что перепутали порядок вызова процедур.

Сначала для простоты отладим “пустую” программу без разбивки пейзажа, и если она работает правильно, тогда составим “пустую” программу полностью. Вот программа без разбивки пейзажа:

```
USES Graph,CRT;
VAR Device, Mode :Integer;

PROCEDURE  Landscape;
   BEGIN  WriteLn('Работает процедура Пейзаж')              END;
PROCEDURE   Music;
   BEGIN  WriteLn('Работает процедура Музыка')              END;
PROCEDURE Flying_Saucer;
   BEGIN WriteLn('Работает процедура Летающая тарелка')  END;

BEGIN
   Device:=0;
   InitGraph(Device,   Mode,   '<путь к гр.др.>');
   DirectVideo:=false;
   Landscape;                                  { рисование пейзажа }
   WriteLn('Работает пауза 3 секунды');        { пауза три секунды }
   Music;                                      { музыка }
   Flying_Saucer;                              { движение тарелки }
   Music;                                      { музыка }
   WriteLn('Работает свет в окне');            { свет в окне }
   ReadLn;
   CloseGraph
END.
```

Пояснение. В нашей программе мы используем и модуль Graph и модуль CRT, так как нам понадобятся и графика и музыка. В этом случае оператор WriteLn просто так не работает. Чтобы он все-таки заработал, необходимо предварительно выполнить оператор *DirectVideo:=false*. Смысл его рассматривать не будем.

В разделе операторов операторы приведены в той последовательности, в которой они должны согласно мультфильму выполняться программой. В разделе описаний процедуры описаны пока в произвольной последовательности. Каждая процедура внутри себя содержит рапорт о своей работе, чтобы мы знали “правильно ли ходят наши трамваи”. Те части программы, которые не удостоились стать процедурами (*пауза, свет*), рапортуют с того места, где в дальнейшем будут записаны на Паскале.

Если все у нас правильно, то результатом работы программы будет такая последовательность сообщений:

```
Работает процедура Пейзаж
Работает пауза 3 секунды
Работает процедура Музыка
Работает процедура Летающая тарелка
Работает процедура Музыка
Работает свет в окне
```

Теперь составим “пустую” программу полностью, с разбивкой пейзажа.

```
USES Graph,CRT;
VAR Device, Mode :Integer;
```

```
PROCEDURE   Tree;
   BEGIN  WriteLn('Работает процедура Дерево')          END;
PROCEDURE   House;
   BEGIN WriteLn('Работает процедура Дом')              END;
PROCEDURE  Landscape;
   BEGIN
      WriteLn('Работает процедура Пейзаж');
      WriteLn('Работает Горизонт процедуры Пейзаж');   { горизонт }
      WriteLn('Работает Синева процедуры Пейзаж');     { синева }
      WriteLn('Работает Луна процедуры Пейзаж');       { луна }
      WriteLn('Работают Звезды процедуры Пейзаж');     { звезды }
      Tree;                                            { дерево }
      Tree;                                            { дерево }
      Tree;                                            { дерево }
      House;                                           { дом }
      House;                                           { дом }
      WriteLn('Работает Трава процедуры Пейзаж');      { трава }
                                                        END;
PROCEDURE   Music;
   BEGIN  WriteLn('Работает процедура Музыка')          END;
PROCEDURE Flying_Saucer;
   BEGIN WriteLn('Работает процедура Летающая тарелка')  END;

BEGIN
   Device:=0;
   InitGraph(Device,  Mode,  '<путь к гр.др.>');
   DirectVideo:=false;
   Landscape;                                  { рисование пейзажа }
   WriteLn('Работает пауза 3 секунды');        { пауза три секунды }
   Music;                                      { музыка }
   Flying_Saucer;                              { движение тарелки }
   Music;                                      { музыка }
   WriteLn('Работает свет в окне');            { свет в окне }
   ReadLn;
   CloseGraph
END.
```

Пояснения: То новое, что появилось по сравнению с укороченным вариантом программы, я выделил жирным шрифтом. Обратите внимание, что раздел операторов совершенно не изменился. Изменилось только содержание процедуры *Landscape* и выше нее появились описания новых процедур, каждая из которых о себе рапортует. (Они описаны именно выше, согласно требованию из 8.2.) Порядок записи операторов внутри процедуры *Landscape* строго соответствует порядку рисования, который я выбрал ранее.

Если наша "пустая" программа верна, то результатом ее работы будет такая последовательность сообщений (новые сообщения я выделил жирным шрифтом):

```
Работает процедура Пейзаж
Работает Горизонт процедуры Пейзаж
Работает Синева процедуры Пейзаж
Работает Луна процедуры Пейзаж
Работают Звезды процедуры Пейзаж
Работает процедура Дерево
Работает процедура Дерево
Работает процедура Дерево
Работает процедура Дом
Работает процедура Дом
Работает Трава процедуры Пейзаж
Работает пауза 3 секунды
Работает процедура Музыка
Работает процедура Летающая тарелка
Работает процедура Музыка
Работает свет в окне
```

Скелет нашей программы готов. Теперь можно наращивать его мышцами операторов. Однако, сначала нужно в программе правильно использовать переменные величины.

10.4. Зачем переменные вместо чисел

В 4.11 я учил вас использовать в программе не числа, а переменные величины. Однако, впоследствие при написании простеньких программ сам же этому принципу не следовал, чтобы не отвлекать ваше внимание на запоминание имен переменных. Теперь пришла пора показать, почему же все-таки полезно вместо чисел употреблять переменные.

Вот программа для задания 3 из 9.7 выше для бесконечного движения окружности влево-вправо. В ней этот принцип пока не соблюдается.

```
USES    Graph;
VAR     x, Device, Mode :Integer;
BEGIN
   Device:=0;
   InitGraph(Device,  Mode,  '<путь к графическим драйверам>');
   ReadLn;
   repeat                    {Внешний цикл для бесконечности отскоков}
                                  { Движение направо:}
      x:=40;                 {Начинаем перемещение из левой части экрана}
      repeat                 {Вложенный цикл для движения направо}
         SetColor(White);
         Circle(x,100,10);   {Рисуем белую окружность}
         SetColor(Black);
         Circle(x,100,10);   {Рисуем черную окружность}
         x:=x+2              {Перемещаемся немного направо}
      until x>600;              {пока не упремся в правый край экрана}
                                  { Движение налево:}
      x:=600;                {Начинаем перемещение из правой части экрана }
      repeat                 {Вложенный цикл для движения налево}
         SetColor(White);
         Circle(x,100,10);   {Рисуем белую окружность}
         SetColor(Black);
         Circle(x,100,10);   {Рисуем черную окружность}
         x:=x-2              {Перемещаемся немного налево}
      until x<40             {пока не упремся в левый край экрана}

   until 8>9                 {Невыполнимое условие, чтобы цикл выполнялся бесконечно}
END.
```

Предположим, мы хотим, чтобы шарик летал в три раза быстрее. Для этого нам достаточно в двух местах программы вместо 2 написать 6. Вот то-то и неудобно, что в двух, а не в одном. Слишком много труда. В нашей программе это, конечно, пустяк, а вот в больших и сложных программах одна и та же величина может встречаться десятки раз, и чтобы ее изменить, придется вносить десятки исправлений.

Теперь напишем вариант той же программы, но с использованием переменных величин:

```
USES    Graph;
VAR     x, Device, Mode, lev, prav, shag     : Integer;
BEGIN
   Device:=0;
   InitGraph(Device,  Mode,  '<путь к графическим драйверам>');
   ReadLn;
   lev:=40;   prav:=600;   shag:=2;
   repeat
      x:=lev;
      repeat
         SetColor(White);
         Circle(x,100,10);
         SetColor(Black);
         Circle(x,100,10);
         x:=x+shag
      until x>prav;

      x:=prav;
      repeat
         SetColor(White);
         Circle(x,100,10);
         SetColor(Black);
```

```
        Circle(x,100,10);
        x:=x-shag
    until x<lev
  until 8>9
END.
```

Теперь для того, чтобы изменить скорость шарика, достаточно заменить 2 на 6 только в одном месте.

Вторая причина, по которой мы используем переменные, та, что с ними программа становится понятнее, так как имена переменным мы придумываем, исходя из их смысла.

10.5. Записываем программу целиком

Пока у вас на экране – правильно работающий «скелет» программы. Наша задача – постепенно наполнить этот скелет мышцами.

Ниже вы можете посмотреть готовую программу мультфильма. А сейчас я поясню порядок ее написания, отладки и отдельные трудные места.

При работе на компьютере ни в коем случае не вводите сразу всю программу из сотни строк, чтобы потом, запустив ее, не схватиться за голову! Почему? Потому что профессиональный программист в среднем допускает одну описку или ошибку в 10 строках. Значит, в вашей программе будет порядка 10 опечаток или ошибок. Все 10 сразу вы не найдете и за сутки. Потому что сразу 10 ошибок искать в 100 раз труднее, чем 10 раз по одной. Хорошо, если вам одновременно врет только один человек. А если вам в уши одновременно врут 10 человек, и все по-разному?

Поэтому, следуя не спеша за материалом этого параграфа, действуйте в таком порядке.

Прочтите часть материала, посвященную горизонту (она чуть ниже). Введите в программу строки, необходимые для рисования горизонта: Прежде всего в раздел VAR *нужно добавить описание* y_goriz. *Затем введите строку* y_goriz:=240. *Затем строку* Line (0, y_goriz, 640, y_goriz). *Запустите программу. Если горизонт на месте и все в порядке, сверьтесь с текстом программы в учебнике и сотрите репортаж горизонта о своей работе.*

Все. С горизонтом разделались. Что дальше должно рисоваться на экране? Синева. Введите пару строк про синеву. Запустите программу. Все в порядке, сверьтесь с текстом. Сотрите репортаж синевы.

И так далее. Вводите за один раз не больше одной-двух строк, после чего запускайте программу и смотрите, как она работает.

Старайтесь не списывать с учебника, а придумывайте программу сами, с учебником только сверяясь.

А теперь приступим к объяснениям в самой программе.

Поскольку у нас первым будет рисоваться горизонт из процедуры Пейзаж, с него и начнем. Я буду исходить из того, что размер вашего экрана - 640×480. Если другой, то вам нетрудно будет внести изменения в текст моей программы.

Пусть горизонт делит экран пополам. Тогда подойдет оператор *Line(0,240,640,240)*. Посмотрим, какие числа здесь целесообразно заменить переменными величинами. Числа 0 и 640 нам менять не понадобится, а вот высоту горизонта мы вполне можем захотеть изменить. Придумаем переменную величину *y_goriz* и поставим ее вместо числа 240. У нас вместо одного оператора получится пара:

```
y_goriz:=240;
Line (0, y_goriz, 640, y_goriz)
```

Оператор *y_goriz:=240* я помещу в начало раздела операторов, так как значение *y_goriz* понадобится и части *Звезды* и процедуре *Летающая тарелка* и мало ли кому еще может понадобиться.

Части *Синева, Луна, Звезды* понятны без пояснений. Попробуем теперь сделать процедуру *Дерево*. Начнем с того дерева, что в левом нижнем углу:

```
PROCEDURE  Tree;
BEGIN
  SetColor(White);
      {Рисуем крону:}
  Ellipse   (100,400,0,360,15,30);
  SetFillStyle(1,LightGreen);
  FloodFill(100,400,White);
      {Рисуем ствол:}
```

```
    Rectangle (97,430,103,460);
    SetFillStyle(1,Brown);
    FloodFill(100,440,White);
END;
```

Сделаем переменными все величины, от которых зависит местоположение дерева, а величины, от которых зависят форма и размеры, для простоты трогать не будем. Будем считать “координатами дерева” координаты центра эллипса, образующего крону. Придумаем им имена *x_tree* и *y_tree* и от них будем отсчитывать все остальные координаты дерева. Тогда процедура будет выглядеть так:

```
PROCEDURE  Tree;
BEGIN
    SetColor(White);
          {Рисуем крону:}
    Ellipse   ( x_tree,  y_tree,   0, 360,15,30);
    SetFillStyle(1,LightGreen);
    FloodFill(x_tree,  y_tree,  White);
          {Рисуем ствол:}
    Rectangle (x_tree - 3, y_tree + 30, x_tree + 3, y_tree + 60);
    SetFillStyle(1,Brown);
    FloodFill  (x_tree,    y_tree + 40,    White);
END;
```

Теперь для того, чтобы дерево было нарисовано в нужном месте, нужно перед выполнением процедуры *Tree* присвоить координатам дерева нужные значения: *x_tree := 100*, *y_tree:= 400*. В программе процедура выполняется три раза, причем каждый раз перед ее выполнением координатам дерева присваиваются разные значения.

Аналогично составляем процедуру *Дом*. Сначала напишем ее без переменных:

```
PROCEDURE  House;
BEGIN
    SetColor(White);
          {Рисуем стену:}
    Rectangle (300,400,340,440);
    SetFillStyle(1,LightBlue);
    FloodFill  (301,401,White);
          {Рисуем окно:}
    Rectangle (315,410,325,420);
    SetFillStyle(1,DarkGray);
    FloodFill  (320,415, White);
          {Рисуем крышу:}
    Line(295,400,345,400);
    Line(295,400,320,370);
    Line(345,400,320,370);
    SetFillStyle(1,Red);
    FloodFill  (320,399, White);
END;
```

Выберем левый верхний угол стены точкой начала отсчета. Ее координаты *x_house:=300*; *y_house:=400*. Окончательный вид процедуры *House* вы можете видеть в программе.

Вот полный текст программы:

```
USES Graph,CRT;
VAR  Device, Mode, i                                  : Integer;
     y_goriz,     x_tree, y_tree,     x_house, y_house,
     x_tar, y_tar,       shirina_tar, visota_tar      : Integer;

PROCEDURE  Tree;
BEGIN
    SetColor(White);
       {Рисуем крону:}
    Ellipse   ( x_tree,  y_tree,     0, 360,    15, 30);
    SetFillStyle(1,LightGreen);
    FloodFill(x_tree,     y_tree,    White);
       {Рисуем ствол:}
```

```
    Rectangle (x_tree - 3,   y_tree + 30,  x_tree + 3,   y_tree + 60);
    SetFillStyle(1,Brown);
    FloodFill  (x_tree,       y_tree + 40,       White);
END;

PROCEDURE   House;
BEGIN
    SetColor(White);
        {Рисуем стену:}
    Rectangle (x_house,       y_house,      x_house+40,     y_house+40);
    SetFillStyle(1,LightBlue);
    FloodFill  (x_house+1,          y_house+1,       White);
        {Рисуем окно:}
    Rectangle (x_house+15, y_house+10,     x_house+25,     y_house+20);
    SetFillStyle(1,DarkGray);
    FloodFill  (x_house+20,   y_house+15,     White);
        {Рисуем крышу:}
    Line(x_house-5,         y_house,      x_house+45,     y_house);
    Line(x_house-5,         y_house,      x_house+20,     y_house-30);
    Line(x_house+45,  y_house,      x_house+20,     y_house-30);
    SetFillStyle(1,Red);
    FloodFill  (x_house+20,   y_house-1,       White);
END;

PROCEDURE  Landscape;
BEGIN
        { горизонт: }
    Line(0,   y_goriz,  640,   y_goriz);
        { синева: }
    SetFillStyle(1,Blue);
    FloodFill(10,10,White);
        { луна: }
    SetColor(Yellow);
    Circle(500,100,30);
    SetFillStyle(1,Yellow);
    FloodFill(500,100,Yellow);
        { звезды: }
    for i:=1 to 100 do    PutPixel  (Random(640),Random(y_goriz),Random(16));
        {Три дерева: }
    x_tree:=100;y_tree:=400;     Tree;
    x_tree:=300;y_tree:=300;     Tree;
    x_tree:=550;y_tree:=380;     Tree;
        {Два дома: }
    x_house:=300;  y_house:=400;       House;
    x_house:=470;  y_house:=300;       House;
        { трава: }
    SetFillStyle(1,Green);
    FloodFill(10,470,White);
END;

PROCEDURE   Music;
BEGIN
    Sound(200);Delay(1000);
    Sound(500);Delay(1000);
    Sound(300);Delay(1000);
    NoSound
END;

PROCEDURE Flying_Saucer;
BEGIN
    {Задаем начальные координаты летающей тарелки и ее размеры.
    Их не обязательно задавать в начале раздела операторов, так как они
    больше никакой процедуре не нужны}
    x_tar:=100;  y_tar:=40;   shirina_tar:=30;  visota_tar:=15;
    repeat                        {Цикл движения тарелки }
        SetColor(White);
```

```
        Ellipse (x_tar, y_tar, 0, 360, shirina_tar, visota_tar);
        Delay(20);                {Чтобы замедлить движение }
        SetColor(Blue);
        Ellipse (x_tar, y_tar, 0, 360, shirina_tar, visota_tar);
        y_tar:=y_tar+1
    until y_tar > y_goriz;
          {Прорисовываем последний раз тарелку на горизонте: }
    SetColor(White);
    Ellipse (x_tar, y_tar, 0, 360, shirina_tar, visota_tar);
END;

BEGIN
    Device:=0;
    InitGraph(Device,   Mode,    '<путь к граф.драйверам>');
    y_goriz:=240;
    Landscape;                                    { рисуем пейзаж }
    Delay(3000);                                  { пауза три секунды }
    Music;                                        { музыка }
    Flying_Saucer;                                { движение тарелки }
    Music;                                        { музыка }
    SetFillStyle(1,Yellow);                       { свет в окне }
    FloodFill  (x_house+20,  y_house+15,    White);    { свет в окне }
    ReadLn;
    CloseGraph
END.
```

10.6. Порядок описания переменных, процедур и других конструкций Паскаля

Если вы помните (4.6), перед тем, как выполниться, программа на Паскале компилируется на машинный язык. При компиляции она просматривается сверху вниз, при этом Паскаль строго следит, чтобы ни одна переменная, процедура или другая конструкция не была в тексте программы применена выше, чем описана. Что имеется в виду?

У нас переменные *описаны* в разделе VAR. А под *применением* переменной будем пока понимать ее упоминание в разделе операторов основной программы или среди операторов в описаниях процедур, то есть там, где эта переменная должна “работать” в процессе выполнения программы.

Посмотрим на нашу программу. Мы мудро поместили раздел VAR на самый верх программы. А если бы мы поместили его, скажем, между описаниями процедур *Music* и *Flying_Saucer*, то например, переменная *i* была бы применена в тексте (в операторе for расположенной выше процедуры *Landscape*) выше, чем описана, на что Паскаль среагировал бы сообщением об ошибке. То же касается и переменной *y_goriz*, которая бы в этом случае была применена два раза ниже описания (в операторах *repeat* и *y_goriz:=240*), но один раз - выше (в операторе *Line(0, y_goriz, 640, y_goriz)*). Не путайте - в данном конкретном случае важен не порядок исполнения операторов в процессе выполнения программы, о котором мы заранее часто и сказать ничего не можем, а примитивный порядок записи описаний и операторов в тексте программы.

Те же рассуждения применимы и к процедурам и другим конструкциям. Так, описание процедуры *Tree* ни в коем случае нельзя было помещать ниже описания процедуры *Landscape*, так как в процедуре *Landscape* процедура *Tree* применяется, причем три раза.

В некоторых случаях, однако, возникает необходимость, чтобы не только процедура, скажем, *P1* обращалась к процедуре *P2*, но и процедура *P2* обращалась к процедуре *P1*. Очевидно, программа должна была БЫ строиться по такой схеме:

```
PROCEDURE P2;              описание процедуры P2
        BEGIN….
          P1 .....         применение процедуры P1
        END;
PROCEDURE P1;              описание процедуры P1
        BEGIN .....
          P2 .....         применение процедуры P2
        END;
```

```
BEGIN .....
        P1 .....                применение процедуры P1
END.
```

Но эта схема противоречит упомянутому принципу, так как применение процедуры *P1* предшествует ее описанию. В Паскале существует способ справиться с этой ситуацией. Достаточно полный заголовок процедуры *P1* скопировать в любое место выше описания процедуры *P2*, снабдив его так называемой **директивой FORWARD**. Программа примет такой вид:

```
PROCEDURE P1; forward;      опережающее описание процедуры P1
PROCEDURE P2;               описание процедуры P2
        BEGIN….
          P1 .....          применение процедуры P1
        END;
PROCEDURE P1;               описание процедуры P1
        BEGIN .....
          P2 .....          применение процедуры P2
        END;
BEGIN .....
        P1 .....            применение процедуры P1
END.
```

10.7. Управление компьютером с клавиатуры. Функции ReadKey и KeyPressed

Попробуйте запустить программу, которая долго делает свое дело, не обращая на вас внимания. Например, такую:

BEGIN repeat WriteLn('А нам все равно!') until 2>3 END.

Вы сидите перед компьютером и ждете, когда он закончит печатать свой текст. А он никогда не закончит. Вы принимаетесь стучать по клавишам, надеясь, что это прервет бессмысленный цикл. Бесполезно.

Только когда вы, удерживая нажатой клавише *Ctrl*, щелкнете по клавише *Break*, программа прервет свою работу.

Пока программы работают, они не реагируют на клавиатуру, если вы об этом специально не позаботились. А чтобы позаботиться, вы должны включить в них специальные функции ReadKey и KeyPressed из модуля CRT. О смысле функций вообще мы поговорим в 13.2, а сейчас разберем на примерах эти две.

Дополним нашу упрямую программу парой строк:

```
USES CRT;
BEGIN
   repeat
      if  KeyPressed then  WriteLn('Хозяин нажал клавишу!')
                   else    WriteLn('А нам все равно!')
   until 2>3
END.
```

Выражение "*if KeyPressed then*" можно перевести как "если нажата клавиша, то". Наткнувшись на это выражение, Паскаль проверяет, была ли нажата клавиша на клавиатуре. Когда вы запустите эту программу, она будет бесконечно печатать *А нам все равно!* Но как только вы щелкнете по какой-нибудь клавише, программа станет бесконечно печатать *Хозяин нажал клавишу!*

Если функция KeyPressed просто реагирует на то, была ли нажата какая-нибудь клавиша, то функция ReadKey сообщает, какая именно клавиша была нажата.

Вот программа, которая бесконечно печатает текст *А нам все равно!* и одновременно непрерывно ждет нажатия на клавиши, и как только клавиша нажата, однократно докладывает, была ли нажата клавиша "*w*" или другая клавиша, после чего продолжает печатать *А нам все равно!* Если мы захотим, чтобы программа закончила работу, мы должны нажать на клавишу "*q*".

```
USES CRT;
VAR   klavisha  : Char;
BEGIN
   repeat
```

```
    Delay (1000);                          {иначе программа печатает слишком быстро}
    WriteLn('А нам все равно!');
    if KeyPressed then begin
       klavisha:= ReadKey;
       if klavisha='w' then WriteLn('Нажата клавиша w')
                       else WriteLn('Нажата другая клавиша')
    end {if}
  until klavisha='q'
END.
```

Программа доберется до строки *klavisha:= ReadKey* только в случае, если будет нажата какая-нибудь клавиша. Функция ReadKey определяет, какой символ был на нажатой клавише, и присваивает его значение переменной, которую мы придумали - *klavisha*. С этого момента вы можете как хотите анализировать эту переменную и в зависимости от ее значения управлять работой компьютера.

Наша программа будет бесконечно печатать *А нам все равно!*, а при каждом нажатии на клавишу будет однократно сообщать *Нажата клавиша w* или *Нажата другая клавиша*. Почему однократно, а не бесконечно? Грубо это можно объяснить тем, что после выполнения функции ReadKey Паскаль "забывает", что на клавиатуре была нажата клавиша.

Вы спросите, а зачем здесь вообще нужна строка *if KeyPressed then*? Дело в том, что если перед выполнением функции ReadKey клавишу на клавиатуре не нажать, то функция ReadKey останавливает программу и заставляет ее ждать нажатия на клавишу, а после нажатия программа продолжает работу. Если вам в вашей программе паузы не нужны, то вам придется использовать KeyPressed.

Функция ReadKey напоминает процедуру ReadLn. Однако у нее есть интересное отличие: при вводе символов по процедуре ReadLn они появляются на экране, а при вводе символа по функции ReadKey - нет. Благодаря этому свойству, с помощью ReadKey можно организовать "секретный ввод" информации в компьютер - человек, стоящий у вас за спиной и видящий экран монитора, но не видящий ваших пальцев, никогда не догадается, на какие клавиши вы нажимаете.

Подробнее о механизме действия ReadKey и KeyPressed см. в следующем параграфе.

Задача "Пароль на программу".

Пусть вы сделали какую-нибудь интересную программу и не хотите, чтобы ее запускал кто угодно. Для этого сделаем так, чтобы программа начинала свою работу с предложения пользователю ввести пароль. Если пользователь набрал правильный пароль, то программа нормально продолжает свою работу, в противном случае прерывается.

Пусть ваш пароль - *typ*. Тогда для решения задачи вам достаточно вставить в начало вашей программы следующий фрагмент:

```
WriteLn('Введите, пожалуйста, пароль');
Simvol1:= ReadKey;
Simvol2:= ReadKey;
Simvol3:= ReadKey;
if NOT ((Simvol1='t') AND (Simvol2='y') AND (Simvol3='p')) then Halt;
{Продолжение программы}
```

Вы скажете: Кто угодно перед запуском моей программы посмотрит в ее текст и сразу же увидит пароль. Совершенно верно. Чтобы текст программы не был виден, преобразуйте ее в исполнимый файл с расширением exe (см. часть IV).

Задание 96 "Светофор": Нарисуйте светофор: прямоугольник и три окружности. При нажатии нужной клавиши светофор должен загораться нужным светом.

Задание 97 "Зенитка": Вверху справа налево медленно движется вражеский самолет (эллипс). В подходящий момент вы нажатием любой клавиши запускаете снизу вверх зенитный снаряд (другой эллипс).

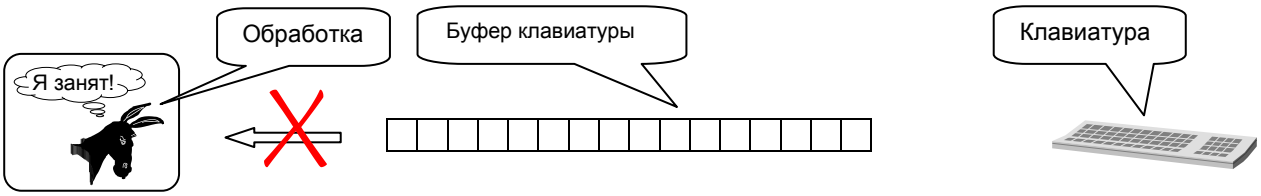
10.8. Буфер клавиатуры

При первом прочтении этот параграф можно пропустить.

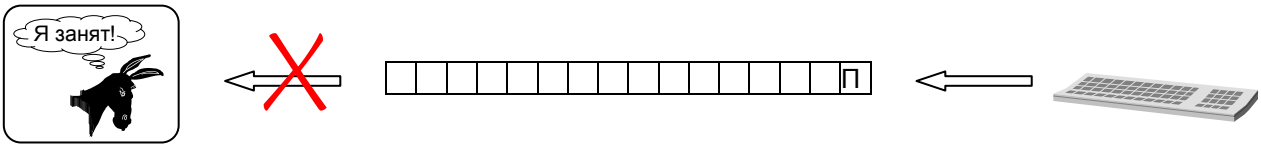
Компьютер работает с клавиатурой сложнее, чем нам кажется. Причина сложности вот в чем. Предположим, вы играете в игру, где с клавиатуры управляете движением самолета. Чтобы избежать попадания вражеского снаряда, вы должны бросить самолет вверх и направо, то есть очень быстро нажать клавишу ↑ и сразу затем клавишу →. Как только вы нажали на ↑, компьютер несколько долей секунды обрабатывает это нажатие, то есть соображает, что теперь ваш самолет нужно бросить вверх, и наконец действительно так и делает. Если вы успели нажать на клавишу → до того, как он это сообразил, то компьютер, не имеющий буфера клавиатуры, просто не обратит внимания на это нажатие, потому что занят мыслительной работой. А это плохо, так как вы совершенно не обязаны во время игры думать о том, успевает ли компьютер за вашими пальцами.

Чтобы исправить ситуацию, нужно дать компьютеру возможность в случае занятости не игнорировать нажатия на клавиши, а запоминать их, чтобы обработать, как только освободится. Для этого и служит **буфер клавиатуры** - место в оперативной памяти, в котором и запоминаются эти нажатия. Вы можете, пока компьютер занят, нажать на клавиши до 16 раз - и буфер клавиатуры запомнит все 16 клавиш в том порядке, в котором они нажимались. Вот как можно изобразить процесс ввода в компьютер текста «Привет недоверчивым!», когда процессор занят:

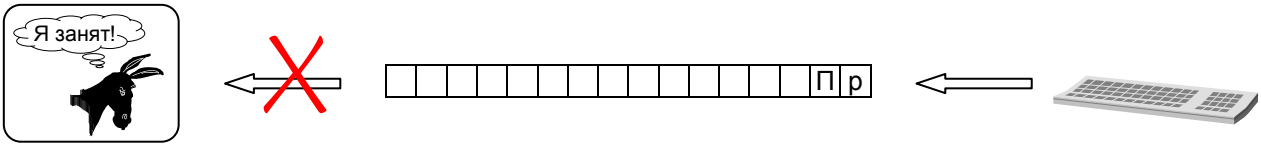
На клавиши пока не нажимали:



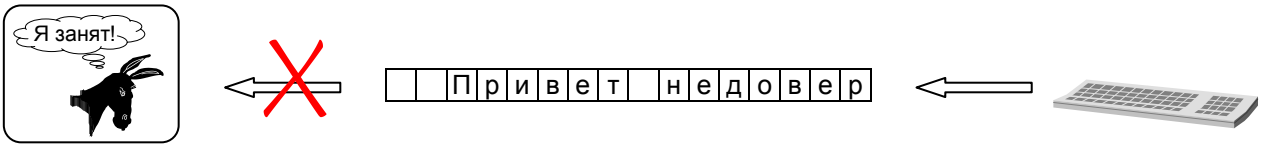
Нажали на клавишу *П*:



Нажали еще на одну клавишу - *р*:



Нажали еще на несколько клавиш:

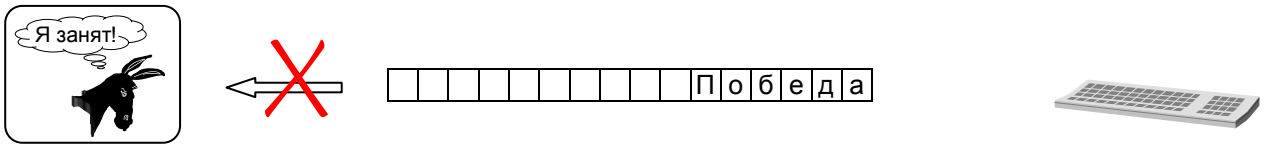


Нажали на клавиши в 15-й и 16-й раз:

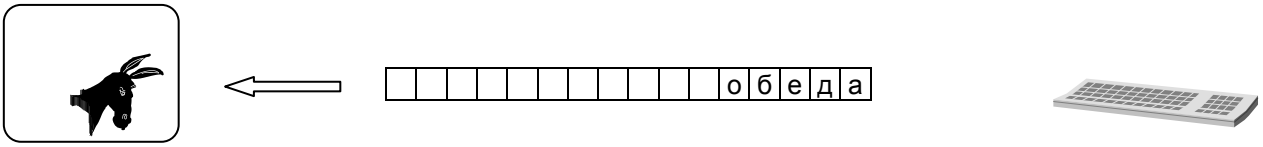
Нажали на клавишу в 17-й раз – раздается предупреждающий писк компьютера, буква *в* в буфер не записывается:

Пока мы буфер клавиатуры только заполняли. А кто его опорожнит? Процессор. Процессор, выполняющий программу на Паскале, берет что-то из буфера клавиатуры только в тот момент, когда выполняет процедуру ReadLn, функцию ReadKey и еще кое-что. В остальное время он для буфера клавиатуры занят. Посмотрим, как он берет информацию из буфера, выполняя ReadKey.

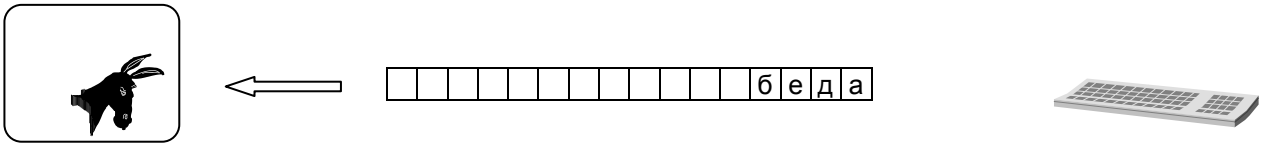
Пусть перед выполнением ReadKey в буфере была такая информация:



При выполнении ReadKey первая из введенных в буфер букв – *П* – отправляется на обработку:



Еще одно выполнение ReadKey:



Помните, что если вы нажали на какую-нибудь клавишу и не отпускаете ее, то это равносильно частому-частому нажатию на эту клавишу. Если процессор в это время не занят выполнением процедуры ReadLn или большого количества функций ReadKey, то некому выуживать информацию из буфера клавиатуры, он мгновенно переполняется и вы слышите раздраженный писк.

На практике события, описанные всеми этими схемами, встречаются редко. Только неопытный пользователь будет жать на клавиши в тот момент, когда компьютер не готов воспринимать информацию. Обычно подавляющую часть времени буфер клавиатуры пуст, ни процессор, ни клавиатура с ним не работают. Раз в вечность человек в нужный момент нажимает на клавишу, в буфере появляется символ и тут же процессор при помощи ReadLn или ReadKey выуживает его оттуда и снова надолго буфер пуст.

Теперь я могу описать правила работы KeyPressed и ReadKey как надо:

Функция KeyPressed отвечает на вопрос, есть ли что-нибудь в буфере клавиатуры. В буфере клавиатуры она никаких изменений не производит.

Функция ReadKey забирает из буфера клавиатуры символ, который попал туда раньше других. Если буфер клавиатуры пуст, то ReadKey останавливает компьютер. Ожидание длится до тех пор, пока в буфере не появится символ (в результате нажатия на клавишу). ReadKey сразу же его оттуда забирает и компьютер продолжает работу.

Теперь вам должно быть понятно, зачем мы в одной из предыдущих циклических программ использовали KeyPressed. Без нее ReadKey просто остановила бы компьютер.

ReadLn при своей работе опустошает буфер клавиатуры.

И еще. Мы знаем, что любая информация в компьютере закодирована (см. 3.5). Поэтому, хоть для простоты я везде говорил, что в буфер попадает символ, правильней было бы сказать, что в буфер попадает код символа.

Как очистить буфер клавиатуры, не останавливая компьютер:

```
while KeyPressed  do  kl:=ReadKey
```

то есть «пока в буфере не пусто, таскай оттуда по символу».

Как ждать нажатия на произвольную клавишу:

```
repeat until KeyPressed
```

то есть «повторяй ничегонеделанье, пока не нажмут на клавишу».

Задание 98 "Управляемая точка": Назначьте четыре клавиши. По нажатии одной из них точка по экрану перемещается на некоторый шаг вверх, другой - вниз, третьей - влево, четвертой - вправо.

В 12.11 вы узнаете, как управлять компьютером при помощи клавиш управления курсором и других.

Задание 99: Добавьте еще пару клавиш - одну для увеличения шага, другую - для уменьшения.

10.9. Гетерархия. Задание на игру "Торпедная атака"

При создании мультфильма в 10.2 мы придерживались стиля программирования сверху-вниз, когда процедуры образовывали подчинение, **иерархию**: все "основные" процедуры вызывались из раздела операторов основной программы. Менее "основные" процедуры вызывались из "основных" и т.д.

Однако, для многих задач, особенно связанных с моделированием сложных объектов и искусственного интеллекта, такой метод неестественен. Процедуры здесь часто получаются равноправными и вызываются не из раздела операторов основной программы, а согласно логике задачи вызывают друг друга по мере необходимости. Такая организация общения процедур называется **гетерархией**.

Сейчас я напишу задание на создание программы для игры "Торпедная атака". А затем предложу схему организации процедур этой программы. В учебных целях я выберу гетерархию, хотя в данном случае можно было бы обойтись и иерархией.

Задание 100: Наверху экрана слева направо плывет вражеский корабль. Внизу притаился ваш торпедный аппарат. В подходящий момент времени вы нажимаете клавишу - и торпеда плывет вверх. Если вы попали, то видна вспышка от взрыва, может быть, на мгновение виден и сам взрыв, раздается коротенькая радостная мелодия, на экране - коротенький поздравительный текст, счетчик подбитых кораблей на экране увеличивается на 1. Если не попали, то зрительные и звуковые эффекты - совсем другие. В любом случае увеличивается на 1 счетчик

выпущенных торпед. Когда торпеды у вас кончатся (скажем, их было 10), игра заканчивается. Программа анализирует ваши успехи и в зависимости от них выдает на экран текст, скажем “Мазила!”, если вы не попали ни разу из 10, или “Профессионал!”, если вы попали 8 раз. Затем спрашивает, будете ли вы играть еще.

Схема программы (читайте ее не сверху вниз, а снизу вверх):

USES CRT,Graph;

VAR все переменные опишем именно здесь, а не внутри процедур

ОПЕРЕЖАЮЩИЕ ОПИСАНИЯ процедур

PROCEDURE ZAVERSHENIE_IGRI; Здесь анализируем, насколько успешно стрелял игрок, отмечаем мелодией, цветом и текстом его достижения, затем спрашиваем, будет ли игрок играть еще. Если да, то вызываем процедуру NACHALO, иначе закрываем графический режим и - Halt.

PROCEDURE NE_PORA_LI; Здесь увеличиваем счетчик торпед. Если он>10, то вызываем процедуру ZAVERSHENIE_IGRI, иначе процедуру RISUNOK.

PROCEDURE NEPOPAL; Здесь программируем все эффекты в случае промаха, после чего вызываем процедуру NE_PORA_LI.

PROCEDURE POPAL; Здесь программируем все эффекты в случае попадания, после чего вызываем процедуру NE_PORA_LI.

PROCEDURE ATAKA; Здесь плывут одновременно корабль и торпеда. Затем в зависимости от ситуации вызываются процедуры POPAL или NEPOPAL. Учтите также ситуацию, когда вы просто забыли выстрелить.

PROCEDURE KORABL; Здесь плывет корабль до выстрела, который вызывает процедуру ATAKA.

PROCEDURE RISUNOK; Здесь рисуем береговую линию, указываем на экране имя игрока, счетчики торпед и подбитых кораблей. Затем вызываем процедуру KORABL.

PROCEDURE NACHALO; Здесь устанавливаем в нуль счетчики торпед и подбитых кораблей, спрашиваем имя игрока и делаем все прочее, что нужно делать один раз за всю игру в самом ее начале. Затем прямо из процедуры NACHALO вызываем процедуру RISUNOK.

```
BEGIN
  инициализация графического режима;
  DirectVideo:=false;                 {Чтобы работал WriteLn }
  NACHALO                             {Вот такой короткий раздел операторов.}
END.
```

Помощь (читайте ее только в крайнем случае). Когда вы выстрелите, вы заметите, что корабль стал плыть медленнее. Это происходит потому, что теперь компьютер за каждый шаг корабля должен еще просчитать и изобразить на экране торпеду, а на это нужно время. Увеличьте шаг движения корабля после выстрела или уменьшите паузу *Delay* так, чтобы скорость осталась примерно той же.

Как компьютер определит, попал или не попал? Нужно в тот момент, когда торпеда доплывет до линии движения корабля, сравнить горизонтальные координаты корабля и торпеды, и если они достаточно близки, считать, что попал.

Улучшение. Если у всех кораблей будет одинаковая скорость, то попадать будет слишком просто, а значит и играть неинтересно. Сделайте скорость кораблей случайной. Конечно, не совсем уж (скажем, в условных единицах скорости диапазон от 0 до 10 – это слишком), а в пределах разумного (скажем, от 4 до 8 – это нормально). Причем не нужно менять скорость одного и того же корабля в процессе движения. Пусть она остается постоянной, а то все будет зависеть не от мастерства, а от везения. Различаются скорости только разных кораблей.

Пусть игрок сможет выбирать из нескольких уровней трудности. Трудность удобнее всего увеличивать, уменьшая размеры корабля, то есть требуемую величину близости координат корабля и торпеды при определении попадания.

Еще одно ***задание 101:*** ***«Графический редактор»***. Создайте программу, которая бы, повинуясь нажатию разных клавиш клавиатуры, рисовала бы, расширяла, сжимала, перемещала по экрану, заливала разными цветами прямоугольники (а если вам понравилось, то и эллипсы и линии и прочее). В качестве «печки, от которой танцевать», можете взять решение задания 98.

Мы с вами закончили первый из двух циклов знакомства с Паскалем. Если вам удались «Торпедная атака» или «Графический редактор» и они у вас работают не хуже, чем указано в задании, то у вас должна появиться уверенность, что теперь, умело разбивая программы на небольшие процедуры, вы можете создавать программы любой сложности, а значит цель этого цикла достигнута. Я вас поздравляю - вам присваивается звание “*Программист-любитель III ранга*”!

Часть III. Программирование на Паскале – второй уровень

Если вам кажется, что вы уже все можете, то вы правы и неправы. Правы потому, что вы можете написать сколь угодно большую программу, разбив ее на разумное число процедур. Неправы потому, что ваша способность манипулировать данными в памяти компьютера еще очень ограничена. А без нее вам не поддадутся «умные» задачи. Например, не познакомившись с так называемыми массивами, вы не сможете запрограммировать игру в крестики-нолики или решить задачу о выходе из лабиринта; не освоив работу со строками, символами и файлами, вы не сможете решить задачу о мало-мальски серьезной секретной шифровке и расшифровке сообщений.

Эта часть посвящена тому, чтобы

- Расширить ваши знания о возможностях Паскаля
- Сделать ваши знания о Паскале строгими, иначе вы не сможете исправлять в программах грамматические ошибки, а значит программы у вас работать не будут.

Начнем с наведения порядка в наших знаниях о Паскале.

Глава 11. Алфавит и ключевые слова Паскаля

11.1. Алфавит

У каждого языка - свой алфавит. В русском языке нельзя употреблять латинские буквы, в греческом - китайские иероглифы и т.д., иначе вас мало кто поймет.

В Паскале тоже есть свой алфавит - четкий набор символов, которые вы имеете право употреблять в программе. Ни одного другого символа употреблять нельзя.

Вот алфавит Паскаля:

1) Латинские заглавные (прописные) буквы: A, B, C, D,....., Z.
2) Латинские малые (строчные) буквы: a, b, c, d,....., z.
3) Десять цифр: 0,1,2,3,4,5,6,7,8,9.
4) Символы подчеркивания "_" и пробела " ".
5) Специальные символы: + - * / = < > ( ) [ ] { } . , : ; ' ^ @ $ #
6) В определенных местах программы можно употреблять и все остальные символы, в том числе и русские буквы. Поясню, о каких символах и о каких местах идет речь. Вы знаете (3.5), что для кодирования одного символа в компьютере используется один байт. Ввиду того, что байт состоит из 8 битов, им можно закодировать 256 символов. Все они приведены в так называемой *таблице ASCII* (в 12.11вам будет предложено распечатать эту таблицу). Большинство из них и имеется в виду. Употреблять их можно только в двух местах:
   - В символьных и строковых константах, например, *Slovo:='Чаща'*
   - Внутри комментариев, например, *{ Вот символы ASCII: %F ! \ Л }*

11.2. Ключевые слова

Существует несколько десятков слов, некоторым из которых не рекомендуется, а большинству просто запрещено быть именами. Происходит это потому, что Паскаль использует их для более важных дел. Эти запрещенные слова, а заодно уж и нерекомендованные (хоть это и нестрого) мы будем называть **ключевыми** (их также называют **зарезервированными** или **служебными**). Вот ключевые слова языков Borland Pascal 7.0 для DOS и TurboPascal 7.0 (списки ключевых слов других версий Паскаля мало чем отличаются от приведенного):

absolute	*exports*	*label*	*resident*
AND	*external*	*library*	*set*
array	*far*	*mod*	*shl*
asm	*file*	*near*	*shr*
assembler	*for*	*nil*	*string*
begin	*forward*	*not*	*then*
case	*function*	*object*	*to*
const	*goto*	*of*	*type*
constructor	*if*	*or*	*unit*
destructor	*implementation*	*packed*	*until*
div	*in*	*private*	*uses*
do	*index*	*procedure*	*var*
downto	*inherited*	*program*	*virtual*
else	*inline*	*public*	*while*
end	*interface*	*record*	*with*
export	*interrupt*	*repeat*	*xor*

Таким образом, если вы ненароком придумаете своей переменной имя *asm*, Паскаль укажет вам на ошибку.

11.3. Использование пробела

- Пробелы запрещены внутри имен и ключевых слов.
- Пробелы обязательны между именами и ключевыми словами (если они находятся в одной строке).
- В остальных случаях пробелы несущественны и могут ставиться произвольно.
- Там, где допускается один пробел, допускается и сколько угодно.
- Вместо нажатия на клавишу пробела можно нажимать на клавишу ввода.

Глава 12. Работа с разными типами данных Паскаля

12.1. Список типов

Каждая переменная величина в Паскале должна принадлежать какому-нибудь типу: Integer, Char, String и т.п. Вот список практически всех типов, многие из которых нам еще предстоит пройти. Здесь не учтены только так называемые процедурные типы, которые в этой книге освещаться не будут.

Простые типы

Числовые типы

Целочисленные типы

Byte
ShortInt
Word
Integer
LongInt

Вещественные типы

Real
Single (при наличии или эмуляции матем. сопроцессора)
Double (при наличии или эмуляции матем. сопроцессора)
Extended (при наличии или эмуляции матем. сопроцессора)
Comp (при наличии или эмуляции матем. сопроцессора)

Символьный тип

Char

Логический тип

Boolean

Перечислимый тип

Ограниченный тип (диапазон)

Сложные (структурированные) типы (строятся из простых):

Массивы *array*
Записи *record*
Множества *set*
Строки *String*

Файлы

Текстовые файлы

Text

Типизированные файлы

File of …

Бестиповые файлы

File

Объекты *Object*

Ссылки:

Ссылки

Адресный тип

Pointer

Комментарии к списку типов

Переменная **простого типа** в каждый момент времени имеет своим значением что-то одно: одно число или один символ и т.п.

Переменная **сложного типа** состоит из многих элементов, каждый из которых принадлежит простому или сложному типу. Элемент сложного типа в свою очередь раскладывается на другие элементы и так далее. Разложив все до конца, мы получим, что переменная сложного типа состоит из элементов простых типов.

Аналогия: *Простой тип*. Вы хотите купить продукты. Вы идете по улице и видите дверь с надписью «Лавка». Вы открываете дверь и видите, что там продаются, например, баранки.

Структурированный тип. Вы хотите купить продукты. Вы видите дверь с надписью «Супермаркет». Вы открываете дверь и видите, что там продается множество разных продуктов.

Ссылки. Вы хотите купить продукты. Вы видите дверь с надписью «Адресное бюро». Вы открываете дверь, но внутри никаких продуктов нет. Вместо них вам дают адрес лавки или супермаркета.

12.2. Числовые типы

Целочисленные типы позволяют переменной принимать значения целых чисел согласно следующей таблице:

Тип	Диапазон значений	Сколько байтов занимает одно значение
Byte	0..255	1
ShortInt	-128..127	1
Word	0..65535	2
Integer	-32768..32767	2
LongInt	-2147483648..2147483647	4

Зачем нужны *Byte* и *ShortInt*, если есть *Integer*? Они занимают меньше места в памяти. Если, например, ваша переменная по смыслу задачи обозначает минуты (то есть целое число в диапазоне от 0 до 60), то полный смысл придать ей тип *Byte*.

Вещественные типы позволяют переменной принимать значения вещественных чисел согласно следующей таблице:

Тип	Примерный диапазон значений	Точность (значащих цифр)	Сколько байтов занимает одно значение
Real	2.9×10^{-39} - 1.7×10^{38}	11-12	6
Single	1.5×10^{-45} - 3.4×10^{38}	7-8	4
Double	5×10^{-324} - 1.7×10^{308}	15-16	8
Extended	3.4×10^{-4932} - 1.1×10^{4932}	19-20	10
Comp [9]	примерно от -10^{19} до 10^{19}		8

Типы *Single*, *Double*, *Extended* и *Comp* могут потребовать для своей работы некоторой настройки Паскаля.

Следует помнить, что дробные числа (например, 1/3) компьютер хранит примерно в таком виде: 0,3333333333333. Вы знаете, что такое представление дробных чисел приблизительно. Чтобы точно представить 1/3, компьютеру понадобилось бы бесконечное количество троек, но память компьютера ограничена. Ячейка под переменную типа *Real* позволяет хранить всего 11-12 таких троек. Эту приблизительность нужно иметь в виду, когда вы многократно выполняете арифметические действия над переменными вещественных типов. При определенном сочетании чисел и действий вы можете немедленно получить совершенно неправильный результат. Попробуйте, например, выполнить такую программу:

```
VAR a,b,y    : Real;
BEGIN
   a:=55555555555.1;   b:=55555555555.0;
   y:=a-b;
   WriteLn (y   :30:3)
END.
```

Вы обнаружите, что вместо результата 0.100 компьютер выдает результат 0.125.

[9] Тип Comp, несмотря на то, что является вещественным, принимает значения только целых чисел.

12.3. Массивы

Для того, чтобы понять массивы, нужно обладать некоторой культурой математического мышления. Если этот материал покажется вам трудным, не поддавайтесь искушению пропустить его. Настоящего программирования без массивов не бывает, да и большая часть дальнейшего материала без массивов не будет понятна.

12.3.1. Переменные с индексами

В математике широко применяются так называемые **индексированные переменные**. На бумаге они записываются так:

x_1 x_2 b_8 y_i y_{i-6} z_{ij} z_{i+1j}

а читаются так: икс первое, икс второе, бэ восьмое, игрек итое, игрек и минус шестое, зет итое житое, зет и плюс первое житое. Поскольку в алфавите Паскаля нет подстрочных букв и цифр, то те же индексированные переменные на Паскале приходится обозначать так:

X[1] X[2] B[8] Y[i] Y[i-6] Z[i,j] Z[i+1,j]

Зачем нужны индексированные переменные? Их удобно применять хотя бы при операциях над числовыми рядами. Числовой ряд – это просто несколько чисел, выстроенных по порядку одно за другим. Чисел в ряду может быть много и даже бесконечно много.

Возьмем, например, бесконечный ряд *чисел Фибоначчи*: 1 1 2 3 5 8 13 21 34..... Попробуйте догадаться, по какому закону образуются эти числа. Если вы сами не догадались, то я подскажу: каждое из чисел, начиная с третьего, является суммой двух предыдущих. А теперь попробуем записать это утверждение с помощью языка математики. Для этого обозначим каждое из чисел Фибоначчи индексированной переменной таким образом:

Первое число Фибоначчи обозначим так: f[1],
Второе число Фибоначчи обозначим так: f[2] и т.д.

Тогда можно записать, что f[1]=1 f[2]=1 f[3]=2 f[4]=3 f[5]=5 f[6]=8 .....
Очевидно, что f[3]=f[1]+f[2],
f[4]=f[2]+f[3],
f[5]=f[3]+f[4] и т.д.

Как математически одной формулой записать тот факт, что каждое из чисел является суммой двух предыдущих? Математики в индексном виде это записывают так:

f[i]=f[i-2]+f[i-1].

Для иллюстрации подставим вместо i любое число, например, 6. Тогда получится:

f[6]=f[6-2]+f[6-1] или
f[6]=f[4]+f[5].

Задание 102: Запишите в индексном виде, как получается из предыдущего числа ряда последующее:

1) 14 18 22 26 .....
2) 6 12 24 48 ....
3) 3 5 9 17 33 65 ....

Вот еще примеры, когда математики предпочитают использовать индексы. Пусть мы на протяжении года каждый день раз в сутки измеряли температуру за окном Тогда вполне естественно обозначить через *t[1]* температуру первого дня года, *t[2]* - второго,....., *t[365]* - последнего. Пусть 35 спортсменов прыгали в высоту. Тогда через *h[1]* можно обозначить высоту, взятую первым прыгуном, *h[2]* - вторым и т.д.

12.3.2. Одномерные массивы

Одна из типичных задач программирования формулируется примерно так. Имеется большое количество данных, например, тех же температур или высот. С этими данными компьютер должен что-нибудь сделать, например, вычислить среднегодовую температуру, количество морозных дней, максимальную взятую высоту и т.п. Раньше мы вычисляли подобные вещи, и данные вводили в компьютер с клавиатуры одно за другим в одну и ту же ячейку памяти (см. 7.7). Однако, программистская практика показывает, что удобно, а часто и необходимо иметь данные в оперативной памяти сразу все, а не по очереди. Тогда для задачи про температуру нам понадобится 365 ячеек. Эти 365 ячеек мы и назовем массивом. Итак, **массивом** можно назвать ряд ячеек памяти, отведенных для хранения значений индексированной переменной. Вопрос о том, как большое количество значений оказывается в памяти, отложим на будущее (15.1).

Рассмотрим на простом примере, как Паскаль управляется с массивами. Предположим, в зоопарке живут три удава. Известна длина каждого удава в сантиметрах (500, 400 и 600). Какая длина получится у трех удавов, вытянутых в линию?

Обозначим длину первого удава - *dlina[1]*, второго - *dlina[2]*, третьего - *dlina[3]*. Прикажем Паскалю отвести под эту индексированную переменную массив:

```
VAR  dlina :  array [1..3]  of Integer
```

Здесь **array** означает массив или ряд, 1 - первое значение индекса, 3 - последнее. Две точки обозначают диапазон от 1 до 3 (см. 5.7) В целом эту строку можно перевести так: Отвести в памяти под переменную dlina ряд ячеек типа Integer, пронумерованных от 1 до 3.

Вот программа полностью:

```
VAR  dlina     :array [1..3]  of Integer;
     summa   :Integer;
BEGIN
   dlina[1]:=500;
   dlina[2]:=400;
   dlina[3]:=600;
      {В этот момент в трех ячейках памяти уже находятся числа
      и с ними можно выполнять арифметические действия}
   summa:= dlina[1]+dlina[2]+dlina[3];
   WriteLn(summa)
END.
```

Если смысл написанного выше вам неясен, запустите отладочный пошаговый режим выполнения программы, заставив Паскаль показывать вам текущие значения *dlina[1], dlina[2], dlina[3], summa*.

Теперь запишем ту же программу в предположении, что длины удавов заранее неизвестны и мы их вводим при помощи ReadLn:

```
VAR  dlina     :array [1..3]  of Integer;
     summa   :Integer;
BEGIN
   ReadLn (dlina[1],dlina[2],dlina[3]);
   summa:= dlina[1]+dlina[2]+dlina[3];
   WriteLn(summa)
END.
```

Теперь решим ту же задачу в предположении, что удавов не три, а тысяча:

```
VAR  dlina     :array [1..1000]  of Integer;
     summa, i :Integer;
BEGIN
      {Вводим длины тысячи удавов, хоть это и утомительно:}
   for i:=1 to 1000 do ReadLn (dlina[i]);
      {Здесь на первом выполнении цикла i=1 и поэтому компьютер выполняет ReadLn(dlina[1]),
      на втором –  i=2 и поэтому компьютер выполняет ReadLn(dlina[2]) и т.д.}

      {Определяем суммарную длину тысячи удавов:}
   summa:= 0;
   for i:=1 to 1000 do summa:=summa+dlina[i]);
   WriteLn(summa)
END.
```

Решим еще одну задачу. Дан ряд из 10 произвольных чисел: *a[1], a[2], ... , a[10]*. Подсчитать и напечатать суммы троек стоящих рядом чисел: *a[1]+a[2]+a[3]*, *a[2]+a[3]+a[4]*, *a[3]+a[4]+a[5]*, ... , *a[8]+a[9]+a[10]*.

```
VAR  a   :array [1..10]  of Integer;
     i    :Integer;
BEGIN
   for i:=1 to 10 do ReadLn (a[i]);
   for i:=1 to 8 do  WriteLn ( a[i]+ a[i+1]+ a[i+2] )
END.
```

Задание 103:. Напишите программу вычисления среднегодовой температуры (Для проверки в компьютере годом можно считать неделю).

Задание 104:. Подсчитайте количество морозных дней (когда температура ниже -20 град.).

Задание 105:. Каким по порядку идет самый морозный день?

Задание 106:. Вычислить и распечатать первые тридцать чисел Фибоначчи.

12.3.3. Двумерные массивы

Поясним суть двумерных массивов на простом примере. Пусть на целом ряде метеостанций, расположенных в разных точках земного шара, в течение многих дней измеряли температуру воздуха. Показания термометров свели в таблицу. Ограничимся для экономии места тремя станциями и четырьмя днями.

	1-й день	2-й день	3-й день	4-й день
Метеостанция 1	-8	-14	-19	-18
Метеостанция 2	25	28	26	20
Метеостанция 3	11	18	20	25

Требуется:

1) Распечатать показания термометров всех метеостанций за 2-й день
2) Определить среднюю температуру на третьей метеостанции
3) Распечатать всю таблицу
4) Распечатать, в какие дни и на каких метеостанциях температура была в диапазоне 24-26 градусов тепла

Для этого обозначим показания термометров индексированной переменной с двумя индексами по следующей схеме:

t[1,1] t[1,2] t[1,3] t[1,4]
t[2,1] t[2,2] t[2,3] t[2,4]
t[3,1] t[3,2] t[3,3] t[3,4]

Обратите внимание, что первый индекс в скобках обозначает номер строки (метеостанции), второй - номер столбца (дня) прямоугольной таблицы.

Программа:

```
{В памяти отводим массив из 3*4=12 ячеек под значения типа Integer индексированной переменной t. Будем называть его двумерным массивом:}

VAR  t      :array [1..3, 1..4] of Integer;
     s,i,j  :Integer;
BEGIN       {Зададим значения элементов массива примитивным присваиванием:}
  t[1,1]:=-8;   t[1,2]:=-14;  t[1,3]:=-19;  t[1,4]:=-18;
  t[2,1]:=25;   t[2,2]:= 28;  t[2,3]:= 26;  t[2,4]:= 20;
  t[3,1]:=11;   t[3,2]:= 18;  t[3,3]:= 20;  t[3,4]:= 25;
            {А теперь распечатаем второй столбец массива:}
  for i:=1 to 3 do WriteLn(t[i,2]);
            {Определим среднее значение элементов третьей строки:}
  i:=3;
  s:=0;
  for j:=1 to 4 do s:=s+t[i,j];
  WriteLn(s/4   :10:3);
            {Распечатаем всю таблицу:}
  for i:=1 to 3 do   for j:=1 to 4 do    WriteLn (t[i,j]);
            {Распечатаем станции и дни с температурой 24-26 градусов:}
  for i:=1 to 3 do   for j:=1 to 4 do
            if (t[i,j]>=24) AND (t[i,j]<=26) then  WriteLn ('Станция ',i,'   день  ',j)
END.
```

Задание 107: Вычислить разницу между максимальной и минимальной температурой во всей таблице.

12.3.4. Какие бывают массивы

Массивы могут быть одномерные, двумерные, трехмерные, четырехмерные и т.д.:

array [1..10] of Integer	-одномерный массив	10 ячеек
array [1..10, 1..5] of Integer	-двумерный массив	50 ячеек
array [1..10, 1..5, 1..2] of Integer	-трехмерный массив	100 ячеек
array [1..10, 1..5, 1..2, 1..3] of Integer	-четырехмерный массив	300 ячеек

Массивы бывают не только числовые, но и символьные, строковые и прочие. Подходит любой известный нам тип. Например:

```
array [1..50] of Char
```

Это означает, что в каждой из 50 ячеек должно находиться не число, а произвольный символ. Еще один пример:

```
array [1..50] of String
```

Здесь в каждой из 50 ячеек должна находиться строка. Примеры программ с такими массивами мы увидим в 12.13.

Границы индексов в квадратных скобках тоже могут быть разными, например:

```
array [20..60] of Real
```

Здесь под вещественные числа отводится 41 ячейка.

```
array [0..9,  -10..30] of Real
```

Здесь под вещественные числа отводится 10*41=410 ячеек.

Вообще индексы могут быть не только числовыми, но и любыми порядковыми. Например,

```
array ['А'..'Я'] of Real
```

Зачем это нужно, будет ясно в 12.8.

Полная синтаксическая информация о массивах будет приведена в 14.8.

Какая польза от массивов при программировании игр? Вряд ли хоть одну «умную» игру можно запрограммировать без применения массивов. Возьмем хотя бы «крестики-нолики» на поле 3 на 3. Вам придется рисовать на экране большие клетки, а в них – нолики (кружочки) после ваших ходов и крестики (пересекающиеся линии) после ходов компьютера. Но этого недостаточно. Чтобы компьютер мог поставить крестик в свободном поле, он должен хотя бы знать, а в каких клетках крестики и нолики уже стоят. Анализировать для этого информацию о пикселах экрана очень неудобно. Гораздо разумнее заранее организовать *VAR a: array[1..3,1..3] of Byte* и записывать туда в нужные места нолики после ходов человека и единички после ходов компьютера. Сразу же после записи в массив 0 или 1 программа должна рисовать в соответствующем месте экрана кружок или крестик. Мыслить компьютер мог бы при помощи примерно таких операторов – *if (a[1,1]=0) AND (a[1,2]=0) then a[1,3]:=1*. Это очевидный защитный ход компьютера.

12.4. Определения констант

Приведем программу вычисления среднегодовой температуры для задания 1 из 12.3.

```
VAR s,i:Integer;  t:array [1..365] of Integer;
BEGIN
   for i:=1 to 365 do ReadLn(t[i]);
   s:=0;
   for i:=1 to 365 do s:=s+t[i];
   WriteLn(s/365)
END.
```

Пусть нам потребовалось переделать эту программу на вычисление средней недельной температуры. Для этого достаточно везде в тексте программы число (константу) 365 заменить на число (константу) 7. В больших программах одна и та же константа может встречаться десятки раз и подобный процесс может потребовать значительного расхода времени. В 10.4 мы уже сталкивались с такой ситуацией. Там мы нашли выход в том, что вместо константы записывали переменную величину. Но в нашем случае этого не получится, так как Паскаль запрещает задавать границу в описании массива переменной величиной. В таких случаях поступают следующим образом. Константе придумывают имя (как переменной), например k, и в специальном разделе CONST ей задается значение. Вот наша программа с использованием константы k:

```
CONST  k  =365;  {Обратите внимание, что в определении вместо  :=  стоит  = }
VAR    s,i :Integer;
       t   :array [1..k] of Integer;
BEGIN
   for i:=1 to k do ReadLn(t[i]);
   s:=0;
   for i:=1 to k do s:=s+t[i];
   WriteLn(s/k)
END.
```

В приведенном виде программа стала универсальной. Теперь для ее переделки под средненедельную температуру достаточно в одном месте поменять определение $k=365$ на $k=7$.

Значению константы запрещено меняться в процессе выполнения программы, то есть запрещены операторы вида $k:=30$ и $ReadLn(k)$. Паскаль присматривает за этим.

Тип константы указывать нельзя[10]. Паскаль сам догадается о типе по записи:

```
CONST n      =800;        {тип целочисленный}
      e      =2.7;        {тип Real}
      bukva  ='ж';        {тип Char}
      Slash  ='/';        {тип Char}
      slovo  ='хорошо';   {тип String}
      OK     =true;       {тип Boolean}
```

Имя константы образуется из букв и цифр так же, как и имя переменной. Важное отличие константы от переменной в том, что значение переменной задается на этапе выполнения программы, а значение константы - раньше, на этапе компиляции. Рекомендую вам там, где можно, вместо переменных применять константы. Программа получается строже.

[10] если это не типизированная константа (речь о них – в следующем параграфе)

12.5. Типизированные константы

В блоке CONST можно описывать не только константы, но и переменные величины. Эти переменные величины из-за того, что они описаны в таком странном месте, приобретают неудачное название **типизированные константы**, но переменными быть не перестают, а самое для нас главное - здесь им можно удобно придавать начальные значения. Вот пример:

```
CONST n  :Integer      =800;
      e  :Real         =2.7;
```

Вот как запишется программа для вычисления средней недельной температуры из 12.4, если массив *t* описать как типизированную константу:

```
CONST  k   =7;        { k - обычная, нетипизированная константа}
       t   :array [1..k] of Integer = (-14, -12, -8, -2, 0, 1, -3);
                      { t - типизированная константа}
VAR    s,i :Integer;
BEGIN
   s:=0;
   for i:=1 to k do s:=s+t[i];
   WriteLn(s/k)
END.
```

Здесь в круглых скобках задается список начальных значений переменной *t*, а именно: *t[1]* равно -14, *t[2]* равно -12 и т.д. В разделе операторов эти значения можно менять.

Двумерным массивам начальные значения придаются аналогично. Так в программе из 12.3 вместо двенадцати присвоений можно было записать так:

```
CONST  k   =3; n=4;
       t   :array [1..k,1..n] of Integer = (( -8,-14,-19,-18),
                                           ( 25, 28, 26, 20),
                                           ( 11, 18, 20, 25));
.......
```

Обратите внимание на расстановку скобок.

12.6. Придумываем типы данных

Паскаль предоставляет возможность не только пользоваться стандартными типами данных, но также именовать их по-другому и даже создавать свои типы.

```
Запись      TYPE  bukva   =                        Char
означает:   ТИП   bukva   "равен" (эквивалентен) типу   Char,
```

то есть мы просто придумали типу Char еще одно название "bukva". Теперь все равно, как записать:

```
                    VAR a,b:Char
или                 VAR a,b:bukva .
```

```
Еще примеры:  TYPE  Vector     = array[1..10] of Integer;
              matritsa = array[1..8] of Vector;
              VAR   a,b      :Vector;
              c      :matritsa;
              d      :array[1.. 8] of Vector;
```

Здесь мы создали два новых типа с именами Vector и matritsa. Очевидно, переменные *c* и *d* описаны одинаково.

```
Обратите внимание, что вместо   TYPE matritsa = array[1.. 8] of Vector
можно записать                  TYPE matritsa = array[1.. 8] of array[1..10] of Integer
или                             TYPE matritsa = array[1..8,1..10] of Integer .
```

Зачем нужны новые типы? Вот две из нескольких причин. Одна – наглядность и удобство. Другая - чисто грамматическая - Паскаль разрешает в определенных конструкциях записывать лишь имена типов, а не их определения. Например, когда мы изучим процедуры с параметрами, мы узнаем, что

```
писать      PROCEDURE p(a: array[1..10] of Integer)   неправильно,
а писать    PROCEDURE p(a: Vector)                    правильно.
```

12.7. Логический тип Boolean

В операторах if, while, repeat мы привыкли писать выражения вида $a>b$, $i<=0$, $c='крот'$, $3>2$, $(a>b)AND(a>c)$ и т.п. Про каждое из этих выражений можно сказать, истинно оно в данный момент или ложно. Например, выражение $3>2$ истинно всегда, а выражение $i<=0$ ложно в тот момент, когда i равно, скажем, 2. Такие выражения называются **логическими выражениями**.

Говорят, что логическое выражение $3>2$ имеет значение “истина” (по-английски **true** - “тру”), а логическое выражение $i<=0$ имеет значение “ложь” (по-английски **false** - “фолс”).

Внутренняя идеология построения языка Паскаль требует определить новый тип переменных - **логический тип Boolean**. Запись $VAR\ a:Boolean$ означает, что переменная a может принимать всего два значения - **true** и **false**. Так, мы можем записать $a:=false$.

Слова true и false являются **логическими константами** и их можно употреблять в логических выражениях или вместо них. Например, $if\ \ a=true\ \ then...$ Конструкцию $if\ (a>b)=false\ then...$ можно перевести “*если неправда, что a больше b, то...*”.

Значения *true* и *false* удобно применять для организации бесконечных циклов:

while true do .....
repeat ..... until false

Решим конкретный пример на этот тип.

Задача: В группе - 6 студентов. Сколько из них сдали зачет по физике?

Сначала напишем программу без использования типа Boolean. В ней единицей я обозначил зачет, нулем - незачет. Массив *Zachet* из 6 элементов хранит информацию о зачете.

```
CONST  Zachet   :array[1..6] of Integer = (1,1,0,1,1,1);
VAR    c,i      :Integer;
BEGIN
   c:=0;                                  {c - счетчик зачетов}
   for i:=1 to 6 do   if zachet[i] = 1    then c:=c+1;
   WriteLn(c)
END.
```

Теперь напишем программу с использованием типа Boolean. В ней через true я обозначил зачет, через false - незачет.

```
CONST  Zachet   :array[1..6] of Boolean = (true,true, false, true, true, true);
VAR    c,i      :Integer;
BEGIN
   c:=0;
   for i:=1 to 6 do    if zachet[i] = true    then c:=c+1;
   WriteLn(c)
END.
```

Отличие второй программы от первой в том, что выражение $zachet[i]\ =\ true$ (зачет равен истина) выглядит естественнее и понятнее, чем $zachet[i]\ =\ 1$ (зачет равен единице, то есть колу?). В общем, чуть-чуть нагляднее.

Кстати, вполне правильно было бы написать и $if\ zachet[i]\ then\ ....$ Ведь условием после слова if может стоять любое логическое выражение, имеющее значением true или false.

12.8. Перечислимые типы

В 5.7 я говорил о порядковых типах - это те типы, все значения которых можно выстроить по порядку и перечислить от начала до конца. Мы пока знаем, что в Паскале порядковыми типами являются целочисленные типы, символьный тип и логический тип. Кроме того, программист может придумывать собственные порядковые типы. Рассмотрим, например, такую конструкцию:

VAR Month : (january, february, march, april, may, june, july, august, september, october, november, december)

Она означает, что переменная $Month$ может принимать только одно из перечисленных в скобках значений. Например, можно записать $Month:=\ may$. Переменная $Month$ является переменной **перечислимого** типа, который является одним из видов порядковых типов.

Эти значения ни в коем случае не являются строками. Так, нельзя записать $Month:=\ 'may'$. Кроме того, их нельзя вывести на печать, вообще они не могут быть введены в компьютер или выведены из него, например, при помощи операторов Read и Write. Однако, их удобно применять при программировании. Это удобство выяснится из следующего примера.

Задача: Известно, сколько дней в каждом месяце года. Сколько дней летом?

Сначала запишем программу традиционным способом.
Программа:

```
CONST  dni:array[1..12] of Byte = (31,28,31,30,31,30,31,31,30,31,30,31);
VAR    s,i :Integer;
BEGIN
   s:=0;                              {Сумматор летних дней}
   for i:=6 to 8 do   s:=s+dni[i];    {Летние месяцы - 6,7,8}
   WriteLn(s)
END.
```

Недостаток приведенной программы - не самая лучшая наглядность, к тому же приходится самому на пальцах вычислять номера месяцев начала и конца лета (6 и 8). Паскаль имеет средства повысить наглядность и удобство таких программ. Запишем нашу программу по-новому, с использованием перечислимого типа данных:

```
TYPE   mes =(january, february, march, april, may, june, july, august, september,
             october, november, december);
CONST  dni   :array[january..december] of Byte =
                          (31,28,31,30,31,30,31,31,30,31,30,31);
VAR    s     :Integer;
       i     :mes;
BEGIN
   s:=0;
   for i:=june to august do s:=s+dni[i];
   WriteLn(s)
END.
```

Пояснения: Основное достижение нашей программы в том, что в операторе *for* можно написать *june to august* вместо *6 to 8*, а в определении массива *dni* можно написать *array[january..december]* вместо *array[1..12]*. Для этого пришлось определить специальный перечислимый тип *mes*, перечислив в скобках произвольные имена месяцев, а переменную цикла *i* задать типом *mes*, а не *Integer*.

Синтаксис перечислимого типа:

```
(имя , имя , имя , . . . . , имя)
```

Значения перечислимого типа можно использовать так же свободно, как и значения порядковых типов, например:

if i = february then dni[i]:= 29

12.9. Ограниченный тип (диапазон)

Задача: Поезд отправляется в путь в 22 часа и находится в пути 10 часов. Во сколько он прибывает на место назначения?

Ошибочная программа:

```
VAR Otpravlenie, Pribitie :Byte;
BEGIN
   Otpravlenie:=22;
   Pribitie:=Otpravlenie+10;
   WriteLn(Pribitie)
END.
```

Эта программа вместо ответа “8” напечатает ответ “32” и ошибки не заметит. Паскаль не знает, что имеют смысл только те значения переменной *Pribitie*, которые находятся в диапазоне от 0 до 24. Это должен был знать программист, но он тоже не обратил на это внимания. Хотелось бы, чтобы Паскаль вместо выдачи неправильного ответа напоминал забывчивым программистам, что переменная вышла из имеющего смысл диапазона. Для этого программист должен иметь возможность этот диапазон Паскалю указать. Такую возможность дает применение **диапазонов (ограниченных типов)**.

Вот программа, обнаруживающая собственную ошибку:

```
VAR Otpravlenie, Pribitie : 0..24;
BEGIN
   Otpravlenie:=22;
   Pribitie:=Otpravlenie+10;
   WriteLn(Pribitie)
END.
```

Конструкция *VAR Otpravlenie, Pribitie : 0..24* означает, что переменные *Otpravlenie* и *Pribitie* имеют право принимать значения целых чисел в диапазоне от 0 до 24.

Паскаль будет обнаруживать выход за диапазон только в том случае, когда вы установите флажок (крестик) в ***Options→Compiler…→Compiler Options→Runtime Errors*** *в положение* ***Range Checking*** *(см. часть IV – «Обзор популярных команд меню»).*

Диапазон можно задавать для любого порядкового типа, например:

```
VAR  Month:(january, february, march, april, may, june, july, august, september, october, november, december);
   Spring    :march..may;
   Autumn    :september..november;
   tsifra    :0..9;
   Zaglavnie:'A'..'Я'
```

Диапазон является одним из видов порядковых типов.

Задание 108: Известны дата и время (месяц, день, час, минута) отплытия теплохода летом этого года из Москвы в Астрахань. Известно время в пути (в днях, часах и минутах). Оно не превышает 20 суток. Определить дату и время прибытия теплохода в Астрахань. Использовать диапазоны.

Вариант 1: Для простоты предположим, что путешествие начинается между 1 и 10 июня.

Вариант 2. Путешествие начинается в любой день лета. Определите еще и дни недели отправления и прибытия, если известно, какой день недели был 1 июня.

12.10. Действия над порядковыми типами

Напомню, что порядковыми типами данных в Паскале называются следующие типы: целочисленные типы, Boolean, Char, перечислимый тип и диапазон.

Как видите, сюда не входят вещественные типы и String.

Порядковый тип - это такой тип, все значения которого можно перечислить, посчитать с начала до конца. Например, в тип *Byte* входит всего 256 различных значений, а именно все целые числа от 0 до 255. В тип *Integer* входит 65536 значений - целые числа от -32768 до 32767. Тип *Char* тоже порядковый, так как количество различных символов в Паскале ограничено числом 256.

Любой порядковый тип имеет внутреннюю нумерацию. Пусть мы задали тип *TYPE weekday = (mon,tu,we,th,fr)* . Внутри компьютера *mon* будет иметь номер 0, *tu* - номер 1, *we* - 2, *th* - 3, *fr* - 4. Пусть мы задали переменную *VAR a: array[mon..fr] of Real* . Теперь для компьютера запись *a[we]* означает то же, что и запись *a[2]*, если заранее была бы задана переменная *VAR a: array[0..4] of Real* .

Тип *Char* имеет нумерацию от 0 до 255. Внутренний номер символа есть его код по таблице кодировки ASCII. Например, буква *Б* имеет номер (код) 129.

У целочисленных типов (Byte, ShortInt, Word, Integer, LongInt) внутренний номер совпадает с самим числом. Так, число -58 в типе *ShortInt* имеет номер -58.

Внутренний номер элемента диапазона равен внутреннему номеру элемента типа, для которого создан диапазон. Пусть мы для типа *weekday* создали диапазон *TYPE days = we .. fr*. Здесь *we* будет иметь номер 2, а не 0.

Операции над порядковыми типами:

1. **ORD**. Эта функция выдает (или, как еще говорят - возвращает) внутренний номер значения любого порядкового типа. Например:

Ord('Б') возвращает 129
Ord (we) возвращает 2
Ord(-58) возвращает -58

2. В любом порядковом типе выполняются операции сравнения > < >= <= = <>. Например, справедливы неравенства *'ю' < 'я'*, *we > tu* . Это возможно потому, что операции сравнения выполняются фактически не над самими значениями, а над их внутренними номерами.

3. **SUCC** - сокращение от *successor* (следующий по порядку). Эта функция возвращает следующий по порядку элемент любого порядкового типа. Например:

Succ (8) возвращает 9
Succ('Ю') возвращает 'Я'
Succ (we) возвращает th

4. **PRED** - сокращение от *predecessor* - это *successor* "наоборот". PRED возвращает предыдущий элемент любого порядкового типа. Например:

Pred (25) возвращает 24
Pred('д') возвращает 'г'
Pred (tu) возвращает mo

Эти функции, как и любые другие, можно применять в выражениях. Например, оператор *y:=10+Ord(we)+Succ(8)* присвоит переменной *y* значение 21.

В операторе *for* переменная цикла может быть любого порядкового типа, например:

for ch := 'd' to 'h' do ...

Задание 109: Подсчитать, сколько заглавных букв в диапазоне от *Б* до *Ф*.

Задание 110: Правда ли, что сентябрь наступает позже июля?

Задание 111: В кондитерском магазине стоит очередь за Сникерсами. В очереди - Nina, Olga, Alex, Marianna, Ester, Misha, Tolik, Lena, Oleg, Anton, Pankrat, Robocop, Dima, Donatello, Zina, Sveta, Artur, Ramona, Vera, Igor, Ira. Известно, сколько у каждого денег. Спрашивается:

1) Хватит ли у них всех вместе денег на Сникерс (3 рубля) ?
2) Какой по порядку в очереди стоит Лена?
3) Правда ли, что у Панкрата денег больше, чем у Миши?

Указание: для хранения денег организовать массив.

12.11. Символьный тип Char. Работа с символами

С символьным типом *Char* мы познакомились в 5.6. Значением символьной переменной являются символы из таблицы ASCII.

Для работы с символами вам достаточно кроме вышеизложенных знать еще одну функцию - *Chr*. Выполнив оператор *c1:=Chr(69)*, Паскаль присваивает *c1* значение символа, соответствующего номеру 69 по таблице ASCII, т.е. латинского '*E*'.

Задание 112: Угадайте, что напечатает компьютер, выполнив оператор *Write(Chr(Ord(Succ(Pred('+')))))*

Задание 113: Распечатайте часть таблицы ASCII, конкретнее - символы, соответствующие кодам 32-255. Обратите внимание:

1) на символы так называемой **псевдографики**, применяемые для вычерчивания таблиц в текстовом режиме;
2) на разрыв в расположении строчных букв русского алфавита.

Использование клавиш передвижения курсора для управления компьютером с клавиатуры.

В 10.7 мы с вами научились вмешиваться в работу программы нажатием алфавитных и цифровых клавиш. Например, мы можем записать

if ReadKey='R' then…,

подразумевая какие-либо действия в случае, если нажата клавиша *R*. Если вы знаете коды клавиш по таблице ASCII, то вы можете то же самое записать по другому:

if ReadKey=#82 then…,

так как код клавиши *R* равен 82. Этот способ более универсальный, так как коды в буфер клавиатуры посылают и те клавиши клавиатуры, которым не приписано никакого символа. Например, клавиша *Tab* посылает код 9.

Итак, нам хотелось бы для управления компьютером использовать и другие клавиши, например, ← → ↑ ↓. Сложность в том, что в отличие от алфавитных и цифровых клавиш, эти и некоторые другие клавиши и комбинации клавиш посылают в буфер клавиатуры не один код, а два, причем первый из них – ноль. Например, клавиша ↑ посылает в буфер пару *(0,72)*, клавиша *Insert* посылает в буфер пару *(0,82)*. Эта парочка называется **расширенным кодом**. Вот что будет в буфере, если мы нажмем подряд шесть клавиш: *R R R* ↑ *Insert Insert*:

							82	82	82	0	72	0	82	0	82

Если вы помните механику работы буфера клавиатуры, то можете умелым использованием функции ReadKey выудить факт нажатия нужной вам клавиши. Так, если вы хотите определить, была ли нажата *Insert*, то можете записать такой фрагмент:

kl:= ReadKey; if kl =#0 then if ReadKey=#82 then…

При этом компьютер не спутает невинную клавишу *R* с клавишей *Insert*.

Вот коды, которые посылают некоторые клавиши в буфер клавиатуры:

↑	*0 72*	↓	*0 80*	←	*0 75*	→	*0 77*
Page Up	*0 73*	Page Down	*0 81*	Home	*0 71*	End	*0 79*
Insert	*0 82*	Delete	*0 83*	BackSpace	*8*	Esc	*27*
Tab	*9*	Enter	*13*	пробел	*32*	серый +	*43*
F1	*0 59*	F2	*0 60*	F3	*0 61*	F4	*0 62*
F5	*0 63*	F6	*0 64*	F7	*0 65*	F8	*0 66*
F9	*0 67*	F10	*0 68*	F11	*0 133*	F12	*0 134*

12.12. Строковый тип String. Работа со строками

Со строковым типом *String* мы познакомились в 4.14.

Как можно сэкономить память, работая со строками? Если мы напишем *VAR a:String*, то Паскаль отведет под символы строковой переменной *a* 255 байтов. Если мы не собираемся присваивать переменной *b* значений длиннее, например, 20 байтов, то выгодно написать *VAR b:String[20]*. В этом случае под символы переменной *b* в памяти будет отведено 20 байтов.

Теперь разберем функции для работы над строками.

Исходные данные	Операция	Результат	Пояснение
s1:='Мото'; s2:='роллер'	s3:=s1+s2	s3='Мото-роллер'	Операция + над двумя строками просто соединяет две строки в одну
s5:='Мото-роллер'	k:=Pos('рол',s5)	k=5	Функция Pos возвращает позицию, на которой находится строка 'рол' в строке s5
s3:='Мото-роллер'	l:=Length(s3)	l=10	Функция Length (длина) выдает (возвращает) количество символов в строк
s3:='астро-ном'	s4:=Copy(s3,3,4)	s4= ‘трон’	Функция Copy возвращает часть строки длиной 4, начиная с третьего символа
s5:='Коро-бочка';	Delete(s5,4,2)	s5='Коро-чка'	Процедура Delete удаляет из строки s5 два символа, начиная с четвертого
s6:='Рука'; s7:='баш';	Insert(s7,s6,3)	s6='Руба-шка'	Процедура Insert вставляет в строку s6 строку s7, начиная с третьего символа
x:=2.73284	Str(x:4:2,s8)	s8='2.73'	Процедура Str преобразует число в строку. 4:2 – это желаемый формат числа (см. 14.5)
s8='2.73'	Val(s8,x,Osh)	x=2.73	Процедура Val преобразует строку в число. Параметр Osh должен иметь тип Integer. Он имеет смысл при анализе ошибки в преобразовании

Процедура *Str* может вам понадобиться, например, вот в каком случае. Модуль *Graph* имеет возможность печатать на экране большими красивыми шрифтами (см. 15.6). Но так печатает он только строковый тип. А в программе “Торпедная атака” вам может захотеться печатать красивым шрифтом счетчик подбитых кораблей, который у вас описан, как целочисленный. Вот тут и пригодится *Str*. Примеры использования *Str* и *Val* см.в 15.6.

Если задана строка *s:='Банка'*, то считается автоматически заданным массив символов с тем же именем: *s[1]=’Б’, s[2]=’а’, s[3]=’н’, s[4]=’к’, s[5]=’а’*. Тогда после выполнения оператора *s[3]:=’р’* мы получим *s='Барка'*.

Строки можно сравнивать. Условие *s1=s2* считается выполненным, если обе строки абсолютно одинаковы, включая и пробелы. Сравнение идет посимвольно слева направо. Поэтому считается, что *‘панк’ < ‘парк’*, так как первый несовпадающий символ ‘*р*’ имеет больший номер, чем ‘*н*’.

Задание 114: Среди детей встречается игра, заключающаяся в зашифровке своей речи “для секретности” за счет вставки в произносимые слова какого-нибудь словосочетания, например, “*быр*”. Тогда вместо слова “*корова*” будет произнесено “*кобырробырвабыр*”. Составьте программу, которая распечатывает заданную строку, после каждой второй буквы вставляя “*быр*”.

Задание 115: Давайте поставим задачу шифрования текста более серьезно. Имеется строка текста. Требуется написать программу, которая зашифровывала бы ее в другую строку. Способов шифровки вы можете придумать сколько угодно. Попробуйте такой – заменять каждый символ текста символом, следующим по порядку в таблице ASCII. Тогда слово *КОТ* превратится в слово *ЛПУ*. Составьте, пожалуйста, и программу дешифровки. Когда вы познакомитесь с файлами, вы сможете уже зашифровывать и дешифровывать не отдельные строки, а целые тексты. В том числе и ваши паскалевские программы.

12.13. Записи

На вооружении флота 100 подводных лодок. Адмиралу часто приходится решать задачи такого типа: 1)перечислить названия лодок, имеющих скорость, превышающую скорость вражеской подводной лодки Шредер; 2)подсчитать, сколько лодок имеют на вооружении больше 10 торпед; и т.д. Чтобы ускорить решение таких задач, адмирал приказал занести в компьютер сведения обо всех лодках, включая вражеские лодки *Шредер* и *Рокстеди*, а именно: их названия, скорость и число торпед, находящихся на вооружении.

Отвести место в оперативной памяти под указанную информацию о 102 лодках можно двумя способами: 1)с помощью массивов, 2)с помощью **записей**.

Рассмотрим программу, использующую первый способ. В каждом массиве будет 102 элемента, причем элементы 101 и 102 отведены под лодки противника.

```
VAR  nazvanie :array[1..102] of String;   {Место под 102 названия}
     skorost  :array[1..102] of Real;     {Место под 102 скорости}
     torped   :array[1..102] of Byte;     {Место под 102 количества торпед}
     i        :Integer;
BEGIN
   {Здесь каким-нибудь способом заносим в отведенное место всю информацию,
    например, присвоением - nazvanie[1]:='Щука'.... или загрузкой из файла}
            {А теперь решим первую из двух задач:}
   for i:=1 to 100 do   if skorost[i] > skorost [101]  then  WriteLn(nazvanie[i])
END.
```

В памяти компьютера информация размещается в том порядке, в котором она встречается в описаниях:

ЯЧЕЙКИ ДЛЯ ИНФОРМАЦИИ	ИНФОРМАЦИЯ
nazvanie[1]	Щука
nazvanie[2]	Дельфин
.....	.....
nazvanie[101]	Шредер
nazvanie[102]	Рокстеди
skorost[1]	26
skorost[2]	24
.....	.....
skorost[101]	20
skorost[102]	18
torped[1]	6
torped[2]	10
.....	.....
torped[101]	15
torped[102]	22
i	?

Вы видите, что данные об одной лодке разбросаны по памяти.

Рассмотрим второй способ. Иногда бывает удобно, чтобы данные, касающиеся одной лодки, хранились в памяти рядом, вот так:

ЯЧЕЙКИ ДЛЯ ИНФОРМАЦИИ	ИНФОРМАЦИЯ
lodka[1].nazvanie	Щука
lodka[1].skorost	26
lodka[1].torped	6
lodka[2].nazvanie	Дельфин
lodka[2].skorost	14
lodka[2].torped	10
. . . . .	. . . . .
vr .nazvanie	Шредер
vr .skorost	20
vr .torped	15
prot .nazvanie	Рокстеди
prot .skorost	18
prot .torped	22

Выстроенную подобным образом информацию в памяти компьютера часто называют **базой данных**.

Сами по себе массивы не позволяют хранить информацию в таком порядке, для этого нужно использовать записи. **Запись** - это набор данных (**полей**) различных типов, касающийся одного объекта. Например, запись, касающаяся нашей первой лодки, это набор трех полей: название - *Щука* (тип *String*), скорость - 26 (тип *Real*), количество торпед - 6 (тип *Byte*). Точка отделяет имя поля от обозначения записи, содержащей это поле.

Напомним, что в массиве разрешается хранить данные только одного типа.

Прежде чем отводить место в памяти под всю информацию, объясним Паскалю, из чего состоит одна запись, то есть опишем ее, задав специальный тип записи **record** и придумав ему имя, скажем, *podlodka*:

```
TYPE podlodka =     record
                        nazvanie    :String;
                        skorost     :Real;
                        torped      :Byte;
                    end;
.........
```

Тип определен, но место в памяти пока не отведено. Здесь нам, хочешь-не хочешь, придется воспользоваться массивом. При помощи *VAR* отведем место под массив из 100 записей для наших подлодок и отдельное место под две записи для вражеских. Массиву придумаем имя *lodka*.

```
VAR   lodka   :array[1..100] of podlodka;
      vr,prot :podlodka;                    {Записи для двух вражеских лодок}
      i       :Integer;
.........................
```

Как видите, элементами массива могут быть не только отдельные числа, символы или строки, но и такие сложные образования, как записи.

Вот программа целиком:

```
TYPE    podlodka =  record
                      nazvanie :String;
                      skorost  :Real;
                      torped   :Byte;
                    end;
VAR  lodka     :array[1..100] of podlodka;
     vr,prot   :podlodka;
     i         :Integer;
BEGIN  {Здесь задаем значения полям всех записей. Конечно, удобнее это делать
        при помощи типизированных констант (см.следующую программу) или
        файлов  данных, но я использую простое присвоение:}
  lodka[1].nazvanie  :='Щука';
  lodka[1].skorost   :=26;
            . . . . . .
  prot.torped        :=22;
                     {А теперь решаем первую задачу:}
  for i:=1 to 100 do if  lodka[i].skorost > vr.skorost  then  WriteLn (lodka[i].nazvanie)
END.
```

Согласитесь, что при использовании записей текст программы гораздо понятнее.

Теперь запишем нашу программу с использованием типизированных констант, для краткости ограничив наш флот тремя подводными лодками:

```
TYPE    podlodka = record
                     nazvanie    :String;
                     skorost     :Real;
                     torped      :Byte;
                   end;
CONST   lodka    :  array[1..3] of podlodka      =
                     ((nazvanie:'Щука';         skorost:26;  torped: 6),
                      (nazvanie:'Дельфин';      skorost:14;  torped:10),
                      (nazvanie:'Леонардо';     skorost:28;  torped:11));
        vr       :  podlodka                    =
                      (nazvanie:'Шредер';       skorost:20;  torped:15);
        prot     :  podlodka                    =
                      (nazvanie:'Рокстеди';     skorost:18;  torped:22);
VAR     i        :  Integer;
BEGIN
   for i:=1 to 3 do  if lodka[i].skorost > vr.skorost then WriteLn(lodka[i].nazvanie);
END.
```

Здесь вы видите, как правильно придавать начальные значения типизированным константам типа record.

Задание 116: Создайте базу данных о своих родственниках. О каждом родственнике должно быть известно:

- *Имя*
- *Год рождения*
- *Цвет глаз*

Массивы не используйте. Программа должна:

1) Распечатать ваш возраст и цвет глаз
2) Ответить на вопрос – правда ли, что ваш дядя старше тети.

Задание 117: Создайте базу данных о своих однокашниках. О каждом однокашнике должно быть известно:

- *Фамилия*
- *Имя*
- *Пол*
- *Год рождения*

Обязательно используйте массив не меньше, чем из 10 записей. Программа должна:

1) Вычислить средний возраст ваших однокашников
2) Определить, кого среди них больше – дам или кавалеров
3) Ответить на вопрос – есть ли в вашей базе тезки (это нелегко).

12.14. Множества

Множеством в Паскале называется набор значений какого-нибудь порядкового типа, подчиняющийся специфическим правилам, о которых мы поговорим дальше. В программе множество записывается в виде списка этих значений в квадратных скобках. Например, *[7,5,0,4]* или *['п', 'ж', 'л']*. Множество не должно состоять более, чем из 256 элементов и не должно содержать элементов с порядковыми номерами меньше 0 и больше 255.

Если в множестве элемент повторяется, то считается, что он входит туда только один раз. Например, множества *[2,5,2]* и *[2,5]* эквивалентны.

Порядок элементов в множестве не играет роли. Множества *[2,5]* и *[5,2]* эквивалентны.

В описании тип множества задается словами **set of**. Например, конструкция

VAR a : set of Byte

говорит о том, что задана переменная, значением которой может быть любое множество из любого числа элементов типа *Byte*. Так, в некоторый момент процесса выполнения программы значением *a* может быть множество *[210, 3, 92]*, а через пару секунд - *[8, 5, 3, 26, 17]*.

Конструкция *VAR c: set of (april, may, june)* говорит о том, что переменная *c* может иметь значением любое множество из имен *april, may, june*. Например, *[april, june]*.

Конструкция *VAR d: set of 10..18* говорит о том, что переменная *d* может иметь значением любое множество целых чисел из диапазона от 10 до 18.

Над множествами определено несколько операций. Рассмотрим три из них: **объединение** (+), **пересечение** (*) и **разность** (-).

Операция	Результат	Пояснение
[1,4,4,5] **+** [1,2,3,4]	[1,2,3,4,5]	В результирующее множество входят элементы, имеющиеся хотя бы в одном из исходных множеств
[1,4,4,5] ***** [1,2,3,4]	[1,4]	В результирующее множество входят только те элементы, которые имеются в каждом из исходных множеств
[1,2,3,4] **-** [1,3,5]	[2,4]	В результирующее множество входят те элементы "уменьшаемого", которые не встречаются в "вычитаемом"

Операция *[1,2]*[3,4]* будет иметь результатом *[ ]*, то есть **пустое множество**.

Вот **операции сравнения множеств**:

if a = b then ...	Если множества *a* и *b* состоят из одинаковых элементов ...
if a <> b then ...	Если множества *a* и *b* отличаются хотя бы одним элементом ...
if a <= b then ...	Если *a* является **подмножеством** *b*, то есть все элементы *a* являются элементами *b* ...
if a >= b then ...	Если *b* является **подмножеством** *a*, то есть все элементы *b* являются элементами *a* ...

Операция проверки вхождения элемента *E* в множество *a*:

*if E **in** a then ...*

Например, *a:= [1,2,4]; if 2 in a then ...* {Если 2 входит в множество a ....}

К сожалению, Паскаль не желает выводить множества на печать, точно так же, как он не желает печатать перечислимые типы. Поэтому просто так узнать, из каких элементов состоит множество, не удастся. Вот один из обходных путей:

Пусть задано множество *a*, описанное, как *set of Byte*. Будем пробовать уменьшать его на все элементы подряд, от 1 до 255, и каждый раз, когда это удается, распечатывать соответствующее число. Вот подходящий фрагмент, в котором мне понадобится "для транзита" еще одно множество *b*:

```
for i:=1 to 255 do begin
      b:=a-[i];
      if a<>b then begin WriteLn(i); a:=b end
end {for}
```

Вот гораздо более короткий и естественный путь:

```
for i:=0 to 255 do if i in a then WriteLn(i)
```

Я думаю, что работа с множествами Паскаля - любопытное и полезное занятие. Например, она нужна математикам, чтобы проверять свои теоремы. Я проиллюстрирую работу с множествами на простеньком примере:

Медиум загадывает шестерку чисел, каждое в диапазоне от 0 до 10 (числа могут и совпадать). Экстрасенс отгадывает их, называя свою шестерку. Есть ли между шестерками совпадающие числа? Если есть, то распечатать их.

Сначала решим задачу традиционными методами, а именно с применением массивов, а не множеств:

```
CONST  razmer  = 10;  kol  = 6;
VAR    Medium, Extrasens   :array[1..kol] of  0..razmer;
       i, j, k             :Integer;
BEGIN
            {Формируем случайным образом две шестерки:}
  Randomize;
  for i:= 1 to kol do begin
    Medium[i]    :=Random(razmer+1);
    Extrasens[i] :=Random(razmer+1)
  end {for};
            {Проверяем две шестерки на совпадение:}
  k:=0;                          {Нам придется подсчитывать количество совпадений.  k - счетчик}
  for i:= 1 to kol do
    for j:= 1 to kol do
      if Medium[i] = Extrasens[j] then begin
        k:=k+1;
```

```
            WriteLn(Medium[i])          {Распечатываем совпадения}
        end {if};
    if k=0 then WriteLn('Не угадал ни разу')
END.
```

У данной программы есть недостатки. Пусть медиум загадал числа *2 4 1 5 4 8*, а экстрасенс назвал *1 4 9 6 1 4*. Программа распечатает числа *4 4 1 1 4 4*, а достаточно было бы только *1 4*. К тому же пришлось организовывать счетчик совпадающих чисел, чтобы иметь возможность ответить, угадано ли хоть одно число.

А теперь применяем множества:

```
CONST  razmer  = 10;  kol  = 6;
VAR    Medium, Extrasens, a    :set of  0..razmer;
       i                       :Integer;
BEGIN
       {Формируем случайным образом две шестерки:}
   Randomize;
   Medium:=[ ];    Extrasens:=[ ];                  {Начинаем формировать "с нуля", то есть с пустых множеств}
   for i:= 1 to kol do begin
      Medium      := Medium   +  [Random(razmer+1)];  {Наращиваем по одному элементу в множестве медиума}
      Extrasens   := Extrasens+  [Random(razmer+1)]  {Наращиваем по одному элементу в множестве экстрасенса}
   end {for}

   a:= Medium * Extrasens;                  {Множество a содержит совпадающие числа. Вот так – одним махом.}
   if a=[ ]    then WriteLn('Не угадал ни разу')
               else begin
                    WriteLn('Есть совпадения, вот они: ');
                                                {Распечатываем элементы множества a:}
                    for i:=0 to razmer do if i in a then WriteLn(i);
               end {else}
END.
```

Задание 118: Случайным образом формируется небольшое множество заглавных букв русского алфавита. Определить, входит ли в это множество хотя бы одна из букв *М,И,Ф*.

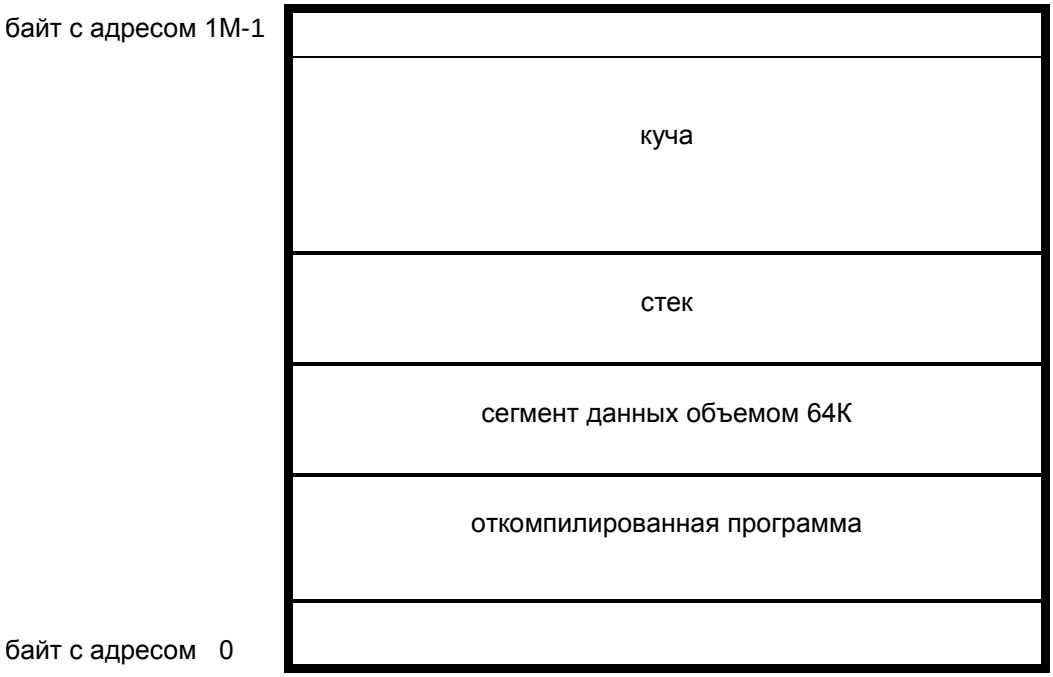
12.15. Расположение информации в оперативной памяти. Адреса

Этот и следующий параграфы носят ознакомительный характер.

Раньше я уподоблял оперативную память тетрадному листу в клеточку. Каждая клетка - байт. Теперь я уподоблю ее многоэтажному небоскребу. Каждый этаж - байт.

Как и положено этажам, байты имеют номера. Эти номера называются **адресами**. Самый "нижний" байт имеет адрес 0, следующий - 1, следующий - - 2 и т.д. Если память вашего компьютера имеет объем 1 Мегабайт, то вы сами можете вычислить адрес последнего байта, учитывая, что 1 Мегабайт = 1024 Килобайта, а 1Килобайт = 1024 байта. Приняты сокращения: Мегабайт - М, Килобайт - К. Имейте в виду, что во многих книгах адреса записываются не в привычном нам виде, а в так называемой шестнадцатеричной системе счисления.

Во время выполнения вашей программы, написанной на Паскале, в памяти находится самая разная информация. То, что относится к паскалевской программе, располагается "по этажам" в следующем порядке:



Границы между некоторыми областями памяти не фиксированы и зависят от решаемой задачи и желания программиста. В сегменте данных располагаются переменные, массивы и другие типы данных вашей программы, описанные привычным вам способом в разделах *VAR, CONST* и т.д. (без использования ссылок). Обратите внимание, что размер сегмента данных весьма невелик (не более 64К). **Стек** - область памяти, в которой располагаются данные, описанные внутри процедур (этого мы пока не делали, об этом - в Глава 13). **Куча** - область памяти, в которой располагаются данные, описанные при помощи ссылок.

12.16. Ссылки

Пусть вы хотите использовать следующий массив:

VAR a: array[1..200, 1..200] of Integer;

Давайте подсчитаем, сколько байтов в памяти займет этот массив. Одно число типа Integer занимает 2 байта. Получаем 200*200*2 = 80000 байтов. В сегменте данных массив не умещается, значит привычным образом работать с ним нельзя. Использование ссылок позволяет разместить его в куче (по английски - heap), имеющей гораздо больший размер.

Я привел лишь один из доводов в пользу применения ссылок. А поближе познакомимся мы со ссылками на простом примере. Задача: Вычислить и напечатать $y=a+b$, где a и b - целые числа 2 и 3. Вот традиционная программа для решения этой задачи:

```
VAR  a, b, y  : Integer;
BEGIN
    a:=2; b:=3;
    y:=a+b;
    WriteLn (y)
END.
```

А теперь потребуем, чтобы число 2 и результат 5 размещались в куче (впрочем, строго говоря, не обязательно в куче). Вот программа со ссылками:

```
VAR  b      :    Integer;
       a,y    :   ^Integer;
BEGIN
    New(a); New(y);
    a^ := 2; b:=3;
    y^ := a^ + b;
    WriteLn (y^)
END.
```

Пояснения: Все, что выше *BEGIN*, выполняется на этапе компиляции: Строка *a,y:^Integer* приказывает отвести в памяти в сегменте данных две ячейки, но не для будущих чисел 2 и 5, а для адресов ячеек из кучи, в

которых эти самые 2 и 5 предполагается хранить. Итак, будущие значения *a* и *y* - не числа 2 и 5, а **адреса** ячеек для этих чисел или, по-другому, **ссылки** на ячейки для этих чисел. Пока же адреса эти не определены.

Все, что ниже *BEGIN*, выполняется на этапе выполнения программы: При помощи обращений к процедуре **New** (*New(a)* и *New(y)*) мы идем дальше и придаем переменным *a* и *y* значения конкретных адресов памяти, то есть отводим для будущих чисел 2 и 5 конкретное место в памяти. Таким образом, мы сталкиваемся с новым для нас явлением - место в памяти отводится не на этапе компиляции, а на этапе выполнения программы. В Паскале имеются средства и освобождать это место на этапе выполнения программы (процедура **Dispose**, на которой я не буду останавливаться). Называется все это **динамическим распределением памяти** и сулит выгоды экономным создателям программ, использующим большие объемы разных данных в разные моменты выполнения программы.

Оператор *a^:= 2* идет еще дальше и посылает в ячейку, адрес которой находится в ячейке *a*, число 2. Обозначается такая ячейка - *a^*. Если бы мне вздумалось написать *a:=2*, это бы значило, что я хочу послать в ячейку *a* адрес равный двум, что вполне возможно, но синтаксически неверно, так как численные значения адресов задаются по-другому.

Смысл следующих двух операторов очевиден.

Подведем итог. Значок ^, поставленный перед типом (например, *^Integer*), означает новый ссылочный тип, значения которого обязаны быть адресами переменной (или ссылками на переменную) исходного типа (в нашем случае *Integer*).

Значок ^, поставленный после переменной ссылочного типа (например, *a^*), означает переменную, на которую ссылается исходная переменная (в нашем случае исходная переменная *a*).

Вот еще некоторые возможные операции со ссылками (без особых пояснений):

```
TYPE    D     = array[1..10] of Real;
        DP    = ^D;
        Int   = ^Integer;
VAR     i, j  :  Int;   { i, j - адреса целых чисел}
        m     :  DP;    { m - адрес первой ячейки массива из 10 вещ. чисел}
BEGIN
   New(i);  New(j);  New(m);
   i^:=4;
   j^:=3;
   j:=i;                {Теперь j и i содержат адреса одного и того же числа - 4}
   WriteLn(j^);         {поэтому будет напечатано число 4}
   m^[9]:=300           {Девятый элемент массива становится равным числу 300}
END.
```

Вернемся к задаче о размещении большого массива. Поскольку Паскаль вслед за MS-DOS запрещает не только описывать, но также, естественно, и ссылаться на структуру, объемом превышающую 64К, то ссылаться сразу на весь двумерный массив не выйдет и поэтому программа получится непростой:

```
TYPE    a     =  array[1..200] of Integer;
        ap    =  ^a;
        a2    =  array[1..200] of ap;
VAR     x     :  a2;          {x - массив из 200 адресов (каждый - ссылка на строку из 200 элементов
                                исходного массива)}

BEGIN
   for i:=1 to 200 do New(x[i]);     {Место для массива отведено}
   .....
   x[128]^[35]:=800;                 {Присвоено значение элементу массива}
   .....
END.
```

В заключение упомяну, что ссылки полезны еще тем, что позволяют организовывать в памяти структуры переменной длины, такие как списки, деревья и т.п.

Глава 13. Процедуры и функции с параметрами

13.1. Процедуры с параметрами

Поставим и решим задачу, данную вам в качестве задания в 8.2: Составьте программу с процедурами, которая исполнит мелодию “Чижик-пыжик” (*ми-до-ми-до-фа-ми-ре-соль-соль-ля-си-до-до-до*).

Воспользуемся нотами третьей октавы:

```
USES CRT;
PROCEDURE doo;   BEGIN  Sound(523); Delay(2000); NoSound  END;
PROCEDURE re;    BEGIN  Sound(587); Delay(2000); NoSound  END;
PROCEDURE mi;    BEGIN  Sound(659); Delay(2000); NoSound  END;
PROCEDURE fa;    BEGIN  Sound(698); Delay(2000); NoSound  END;
PROCEDURE sol;   BEGIN  Sound(784); Delay(2000); NoSound  END;
PROCEDURE la;    BEGIN  Sound(880); Delay(2000); NoSound  END;
PROCEDURE si;    BEGIN  Sound(988); Delay(2000); NoSound  END;
BEGIN
   mi; doo; mi; doo; fa; mi; re; sol; sol; la; si; doo; doo; doo
END.
```

Все процедуры в нашей программе всегда делают одно и то же. Например, процедура *doo* всегда издает звук частоты 523 герца и продолжительности 200 мс. Происходит это потому, что в описании процедур мы использовали не переменные величины, а константы.

В реальной музыке ноты принадлежат разным октавам и имеют разную длительность. Чтобы получить ноту четвертой октавы, достаточно умножить частоту одноименной ноты третьей октавы на 2. Например, нота *ре* четвертой октавы имеет частоту 587*2=1174 гц. Естественно, все ноты четвертой октавы звучат выше нот третьей октавы. Пятая октава получается из четвертой так же, как четвертая из третьей и звучит еще выше. Ноты второй октавы, наоборот, получаются из нот третьей октавы делением частоты на 2.

Поставим задачу - создать более универсальную процедуру. Чтобы заставить ноту звучать по-разному, используем переменные величины. Здесь мы используем ту же самую идею, которую мы использовали в процедуре *House* из Глава 10, она рисовала дом в разных местах экрана в зависимости от значений переменных величин, задающих его координаты. Для простоты ограничимся пока одной нотой *ре* и двумя октавами - третьей и четвертой. Длительность пусть будет любая. Пусть программа должна воспроизвести три подряд ноты *ре*: сначала третья октава одна секунда, затем четвертая октава одна секунда и затем третья октава три секунды.

```
USES CRT;
VAR  octava    : Byte;
     dlit      : Word;
PROCEDURE re;      BEGIN
   if octava = 3  then Sound(587)  else Sound(1174);
   Delay(dlit);
   NoSound         END;
BEGIN
   octava:=3; dlit:=1000;  re;      octava:=4; dlit:=1000;  re;      octava:=3; dlit:=2000;  re
END.
```

Недостаток программы в том, что раздел операторов выглядит довольно мутно. Гораздо прозрачнее была бы такая запись:

```
BEGIN
        re(3,1000); re(4,1000); re(3,2000)
END.
```

Для обеспечения такой прозрачности подходят процедуры с параметрами. Вот программа, использующая процедуру с параметрами:

```
USES CRT;
```

```
PROCEDURE re (octava : Byte; dlit: Word);  BEGIN
   if octava = 3  then Sound(587)  else Sound(1174);
   Delay(dlit);
   NoSound                                  END;
BEGIN
   re(3,1000); re(4,1000); re(3,2000)
END.
```

Пояснения: Эта программа похожа на предыдущую, но имеется несколько отличий. Строка

PROCEDURE re (octava : Byte; dlit: Word)

называется **заголовком процедуры**. Здесь после имени процедуры - *re* - ставятся скобки и внутри них описываются так называемые **формальные параметры** процедуры. Здесь их два: *octava* и *dlit*. Поскольку они описаны в заголовке, пропадает необходимость в разделе *VAR*.

В записи *re(3,1000)* числа 3 и 1000 - так называемые **фактические параметры** процедуры, их порядок и тип должен соответствовать формальным параметрам.

Когда во время выполнения программы Паскаль натыкается на *re(4,1000)*, он присваивает переменной *octava* значение 4, переменной *dlit* - значение 1000 (то есть, присваивает формальным параметрам значения фактических параметров) и затем переходит к выполнению тела процедуры *re*.

Усложним задачу. Создадим универсальную процедуру *nota*, параметрами которой будут название ноты, октава и длительность. Учтем, что длительность ноты в музыке задается в так называемых долях. Доли бывают: *1* - целая нота, *1/2* - половинка длительности, *1/4* - четвертушка и т.д. Пусть целая нота звучит 1 секунду. Вызывать процедуру *nota* можно было бы так: *nota(re,5,8)* - это означает, что мы хотим, чтобы прозвучала нота *re* пятой октавы длительности *1/8*.

Вот запись программы:

```
USES CRT;
TYPE Nota_type = (doo, doo_diez, re, re_diez, mi, fa, fa_diez, sol, sol_diez, la, la_diez, si);

PROCEDURE Nota(Nazvanie:Nota_type; Oktava,Dolya:Byte);   {Здесь параметр Dolya - знаменатель доли}
VAR Hz:Word;      {Внутри процедуры можно описывать свои переменные (в данном примере это Hz).
                   Они называются локальными. Подробнее о них - в 13.3}
BEGIN
                   {Объясним Паскалю частоту нужных нам нот третьей октавы}
   case Nazvanie of
      doo       : Hz:=523;
      re        : Hz:=587;
      sol       : Hz:=784;
      la        : Hz:=880;
      la_diez   : Hz:=932;
   end;
                   {Теперь меняем частоту в зависимости от октавы}
   case Oktava of
      1  : Hz:=Hz div 4;      {Используем целочисленное деление,так как стандартная}
      2  : Hz:=Hz div 2;      {процедура Sound требует задавать частоту целым}
      3  : Hz:=Hz;            {числом герц}
      4  : Hz:=Hz*2;
      5  : Hz:=Hz*4;
      6  : Hz:=Hz*8;
      else WriteLn('Такой октавы не знаю'); ReadLn; Halt
   end;

   Sound (Hz);                {Включаем звук}
   Delay(10000 div Dolya);    {Задаем продолжительность звука}
   NoSound;
   Delay (50);                {Небольшой промежуток тишины после каждой ноты}
END;
BEGIN
                   {Вот первые ноты из песни “Широка страна моя родная”:}
   Nota(re,3,8); Nota(re,3,16); Nota(re,4,4); Nota(re,4,8); Nota(re,4,8); Nota(doo,4,8);
   Nota(la_diez,3,8); Nota(la,3,8); Nota(sol,3,4); Nota(re,3,4)
```

```
END.
```

Фактические параметры могут быть любыми выражениями подходящего типа. Например, вместо *Nota(re,3,8)* можно было бы написать *a:=3; Nota(re, a, 11-a).*

Задание 119: В модуле Graph не хватает процедуры, которая рисовала бы треугольник. Создайте такую процедуру. Она должна рисовать примерно равносторонний треугольник вершиной вверх и иметь три параметра: положение треугольника на экране и размер.

13.2. Функции

В 4.9 мы с вами уже сталкивались со стандартными функциями. Например, выражение *10+Sqr(3)* имеет значение 19, так как функция *Sqr(3)* обозначает 3^2.

Вы можете создавать собственные функции. Предположим, вам часто приходится вычислять периметры прямоугольников. Тогда вам было бы удобно иметь функцию *perimetr(10,4)*, которая имела бы значение периметра прямоугольника со сторонами 10 и 4. Рассмотрим, как это делается, на примере программы вычисления суммарной длины забора вокруг трех несоприкасающихся прямоугольных дворов:

```
FUNCTION perimetr(dlina,shirina:Word) : Integer;
   BEGIN   perimetr:=2*(dlina+shirina)  END;
BEGIN
   WriteLn(perimetr(10,4)+ perimetr(20,30)+ perimetr(3,8));
END.
```

Функции очень похожи на процедуры. Но функция в отличие от процедуры обладает некоторыми свойствами переменной величины и поэтому описание функции отличается от описания процедуры следующими двумя вещами:

- В заголовке функции после скобок с формальными параметрами должен быть указан тип функции (у нас это *Integer*).
- Внутри описания функции между BEGIN и END ей хотя бы раз должно быть присвоено какое-нибудь значение (у нас это *perimetr:=2*(dlina+shirina))*.

Рассмотрим более сложный пример. Вспомним задачу из 12.3 о среднегодовой температуре, где исходными данными является массив из 365 значений температуры. Попробуем узнать, в январе (дни 1-31) или в декабре (дни 335-365) самый теплый день месяца был теплее. Мы можем предвидеть, что для вычисления понадобятся два похожих фрагмента программы, каждый длиной строчки по три-четыре. Чтобы не писать два раза похожие фрагменты, создадим функцию нахождения максимального элемента из заданного диапазона массива температур. Назовем ее *max*. Используя ее, мы можем в программе записать, например, так: *u:=max(20,30)*, подразумевая, что переменной *u* должно быть присвоено значение максимального элемента массива температур, выбирая из элементов с 20-го по 30-й. Вот программа:

```
VAR  t : array [1..365] of Integer;       { t - массив температур за год}
FUNCTION  max (perv,posledn:Word) :Integer;
   VAR i,m :Integer;
   BEGIN
      m:=t[perv];
      for i:=perv+1 to posledn do  if t[i]>m then m:=t[i];
      max:=m
   END;
BEGIN
   ..... {Здесь присваиваем значения элементам массива температур}
   if max(1,31)>max(335,365) then WriteLn('В январе') else WriteLn('В декабре');
END.
```

Задание 120: В Паскале не хватает функции для вычисления произвольной целой степени числа. Создайте функцию *Power* такого смысла: *Power(2,3)* должна иметь значение 2^3, то есть 8.

Задание 121: Если вы никак не можете смириться с системой координат графического режима, то напишите пару простеньких функций (например, с именами *x* и *y*), которые позволят вам считать, что отныне ось *y* направлена вверх, а центр координат расположен в центре экрана. Если вы правильно напишете эти функции, то, например, оператор *Circle (x(310), y(230), 10)* нарисует вам кружочек в правом верхнем углу экрана.

13.3. Подпрограммы. Локальные и глобальные переменные

Будем называть процедуры и функции **подпрограммами**, так как они входят внутрь программы.

Деление переменных на локальные и глобальные является способом повышения надежности больших программ и понижения вероятности запутаться при их написании. Программы, создаваемые сегодня профессиональными программистами, очень велики - десятки и сотни тысяч строк. Таково, например, большинство игровых программ. Естественно, один человек не может достаточно быстро создать такую программу, поэтому пишется она обычно большой группой программистов. Для этого программа делится на десятки и сотни подпрограмм, и каждый программист пишет одну или несколько подпрограмм.

Исследуем взаимодействие подпрограмм в такой программе. Для этого рассмотрим глупую простую "сложную" программу A, вся задача которой - выполнить по порядку следующие вещи:

1) присвоить значение 5 переменной x,
2) затем вызвать процедуру B, зачем-то возводящую 10 в квадрат и печатающую текст "$Результат\ равен$ ",
3) и наконец, напечатать значение x:

```
VAR x,y : Integer;
PROCEDURE B;  BEGIN   y:=10*10;   Write('Результат равен ') END;
BEGIN
   x:=5;
   B;
   WriteLn(x);
END.
```

Очевидно, программа напечатает *Результат равен 5*.

Пусть программу A пишет программист A, он же руководитель всего проекта, а процедуру B - программист B. Когда дело касается большой программы, каждый программист досконально знает только задачу, решаемую его подпрограммой, а об остальных подпрограммах и обо всей программе он может иметь лишь минимальное представление. Задача руководителя проекта - умело разделить программу на подпрограммы и четко поставить задачу для каждой подпрограммы.

Сами подпрограммы тоже достаточно велики, и каждая из них может использовать десятки переменных. При этом возникает опасность, что автор какой-нибудь подпрограммы случайно использует внутри подпрограммы для ее нужд имя переменной, используемой в другой подпрограмме или в общей программе для других нужд, и таким образом испортит ее значение. Пусть, например, в нашей процедуре B ее автор опрометчиво присвоил бы значение 10*10 переменной с именем x. Тогда программа выглядела бы так:

```
VAR x,y : Integer;
PROCEDURE B;  BEGIN   x:=10*10;   Write('Результат равен ') END;
BEGIN
   x:=5;
   B;
   WriteLn(x);
END.
```

Очевидно, данная программа напечатала бы *Результат равен 100*.

Для защиты от таких ошибок руководитель проекта должен внимательно следить, чтобы разные подпрограммы не использовали переменных с одинаковыми именами. Но для больших программ этот контроль очень трудоемок и неудобен. Вместо этого в современных языках программирования разработан механизм локальных переменных. **Локальная** переменная - это переменная, описанная не в главной программе, а внутри подпрограммы. Если программист B знает, что его число 10*10 нигде, кроме как в процедуре B, не нужно, он описывает соответствующую переменную x внутри процедуры B, ничуть не заботясь, что переменные с таким же именем встречаются в других местах программы:

```
VAR x : Integer;
PROCEDURE B;
   VAR x : Integer;
   BEGIN  x:=10*10; Write('Результат равен ')  END;
BEGIN
   x:=5;
   B;
   WriteLn(x);
```

```
END.
```

Данная программа напечатает *Результат равен 5*. Произойдет это вот по какой причине: Переменные, описанные внутри и снаружи подпрограммы, компилятор считает разными переменными, даже если они имеют одинаковые имена. Переменная x, описанная в программе, это совсем другая переменная, чем x, описанная в подпрограмме, и помещаются эти переменные в разных местах памяти. Поэтому и не могут друг друга испортить. Вы можете вообразить, что это переменные с разными именами $x_{глоб}$ и $x_{лок}$.

Переменная, описанная внутри подпрограммы, невидима снаружи. Она *локальна* в этой подпрограмме. Она существует, пока работает подпрограмма, и исчезает при выходе из подпрограммы.

Переменная, описанная в главной программе, называется **глобальной** переменной. Она видна отовсюду в программе и внутри подпрограмм и каждая подпрограмма программы может ее испортить. Но когда подпрограмма натыкается на переменную x, описанную внутри самой этой подпрограммы, то она работает только с ней и не трогает переменную x, описанную снаружи.

Рассмотрим еще один пример:

```
VAR x,z : Integer;
PROCEDURE B;
   VAR x,y : Integer;
   BEGIN
      x:=20; y:=30; z:=40
   END;
BEGIN
   x:=1; z:=2;
   B;
   WriteLn(x,' ',z)
END.
```

Программа напечатает $1\ 40$. Пояснение: Оператор $WriteLn(x,z)$ находится снаружи процедуры B, и поэтому локальная переменная $x=20$, описанная внутри B, из него не видна. Зато прекрасно видна глобальная переменная $x=1$, которую он и печатает. Переменная же z не описана внутри B, поэтому она глобальна, и оператор $z:=40$ с полным правом меняет ее значение с 2 на 40.

Для полной ясности приведу порядок работы компьютера с этой программой:

1) В сегменте данных оперативной памяти компилятор отводит место под X глобальное и Z глобальное.
2) Программа начинает выполняться с присвоения значений X глобальное = 1 и Z глобальное = 2.
3) Вызывается процедура B. При этом в стеке оперативной памяти отводится место под X локальное и Y локальное.
4) Присваиваются значения X локальное = 20, Y локальное = 30 и Z глобальное = 40.
5) Программа выходит из процедуры B. При этом исчезают переменные X локальное = 20 и Y локальное = 30.
6) Компьютер печатает X глобальное = 1 и Z глобальное = 40.

Формальные параметры подпрограмм являются локальными переменными в этих подпрограммах. Сколько их ни меняй внутри подпрограммы – одноименные глобальные переменные не изменятся. Не изменятся также и фактические параметры. В этом смысле они «защищены», что значительно повышает надежность программ. (Это не относится к так называемым параметрам-переменным, о которых речь позже).

13.4. Массивы как параметры

Параметрами подпрограмм могут быть переменные не только простых, но и сложных типов, таких как массивы, записи, множества. Рассмотрим для иллюстрации пример с массивами.

Задача: Имеется два массива, по два числа в каждом. Напечатать сумму элементов каждого массива. Использовать функцию sum, единственным параметром которой является имя суммируемого массива.

Программа:

```
TYPE     vector    =  array [1..2]  of Integer;
VAR      a,b       :  vector;
FUNCTION sum (c:vector):Integer;
   BEGIN sum:=c[1]+c[2] END;
BEGIN
   a[1]:=10;   a[2]:=20;
```

```
    b[1]:=40;   b[2]:=50;
    WriteLn (sum(a),' ',sum(b));
END.
```

Начиная вычислять функцию *sum(a)*, Паскаль подставляет в ячейки для элементов массива *c* значения элементов массива *a*. Начиная же вычислять функцию *sum(b)*, Паскаль подставляет в ячейки для элементов массива *c* значения элементов массива *b*.

В заголовке функции неправильно было бы писать

function sum (c: array [1..2] of Integer):Integer.

Необходимо было сначала определить тип массива в разделе *TYPE*, а затем использовать это определение и в описании *a* и *b*, и в заголовке функции. Таково требование синтаксиса Паскаля.

Задание 122. В школе два класса. В каждом - 5 учеников. Каждый ученик получил отметку на экзамене по физике. Определить, какой из двух классов учится ровнее (будем считать, что ровнее учится тот класс, в котором разница между самой высокой и самой низкой отметкой меньше).

Указание: Создать функции *min(c:vector)*, *max(c:vector)* и *raznitsa(c:vector)*.

13.5. Параметры-значения и параметры-переменные

Многие процедуры не только рисуют или звучат, но и, подобно функциям, вычисляют что-нибудь полезное. Например, процедура *B* из следующей программы увеличивает глобальную переменную *x* на значение параметра *y*.

```
VAR x: Integer;
PROCEDURE B (y:Integer);
   BEGIN  x:=x+y  END;
BEGIN
   x:=1000;
   B(1);
   WriteLn(x)
END.
```

Будет напечатано число 1001.

Однако руководители проектов не любят, когда в подпрограммах встречаются имена глобальных переменных. Мало ли - руководителю придет в голову изменить имя глобальной переменной, и что тогда - переписывать все подпрограммы? Поэтому придумали использовать так называемые параметры-переменные. Вот та же программа с их использованием:

```
VAR x: Integer;
PROCEDURE B (y:Integer;   var c:Integer);
   BEGIN  c:=c+y  END;
BEGIN
   x:=1000;
   B(1, x);
   WriteLn(x)
END.
```

Здесь *y* - хорошо знакомый нам параметр. Называется он **параметр-значение**. При начале выполнения подпрограммы для параметра-значения выделяется место в стеке и туда посылается значение соответствующего фактического параметра (1).

c - незнакомый нам **параметр-переменная**, отличающийся от параметра-значения словом *var*. При начале выполнения подпрограммы для параметра-переменной никакого места в стеке не выделяется, а выделяется в стеке место только для адреса соответствующего фактического параметра. Подпрограмма через этот адрес работает непосредственно с переменной, являющейся фактическим параметром (*x*). Получается, что слово *var* «снимает защиту» со своего фактического параметра и вы вполне можете нечаянно его испортить.

Вопрос: имеет ли смысл писать *B(1, 1000)*? Ответ: не имеет, так как подпрограмма не будет знать, какой переменной присваивать результат 1001. Естественно, Паскаль выдаст сообщение об ошибке.

Задание 123: На двух станциях (*A* и *B*) в течение года измерялась температура. Соответственно созданы два массива чисел длиной 365. Затем оказалось, что на станции *A* термометр все время показывал температуру на 2 градуса выше настоящей, а на станции *B* - на 3 градуса ниже. Написать процедуру с двумя параметрами,

которая исправляет исходный массив. Один формальный параметр - величина поправки, другой - параметр-переменная - массив температур.

13.6. Индукция. Рекурсия. Стек

Начну с классического примера о факториале. Факториалом целого положительного числа N называется произведение всех целых чисел от 1 до N. Например, факториал пяти равен $1*2*3*4*5$, то есть 120. Факториал единицы считается равным 1.

Все понятно. Однако, существует еще один, совершенно ужасный способ объяснения, что такое факториал. Вот он:

“Факториал единицы равен 1*. Факториал любого целого положительного числа* N*, большего единицы, равен числу* N*, умноженному на факториал числа* N-1*.”*

Если вам уже все ясно, значит вы - математический талант. Для нормальных людей поясню. Чтобы последнее предложение было понятнее, возьмем какое-нибудь конкретное N, например, 100. Тогда это предложение будет звучать так: Факториал числа 100 равен числу 100, умноженному на факториал числа 99.

Ну и что? И как же отсюда узнать, чему равен какой-нибудь конкретный факториал, скажем, факториал трех? Будем рассуждать совершенно чудовищным образом:

Смотрю в определение: Факториал трех равен 3 умножить на факториал двух. Не знаю, сколько это. Спускаюсь на ступеньку ниже.

Смотрю в определение: Факториал двух равен 2 умножить на факториал единицы. Не знаю, сколько это. Спускаюсь еще на ступеньку.

Смотрю в определение: Факториал единицы равен 1. Вот - впервые конкретное число. Значит можно подниматься.

Поднимаюсь на одну ступеньку. Факториал двух равен 2 умножить на 1, то есть 2.

Поднимаюсь еще на ступеньку. Факториал трех равен 3 умножить на 2, то есть 6. Задача решена.

Рассуждая таким образом, можно вычислить факториал любого числа. Способ рассуждения называется **рекурсивным**, а способ объяснения называется **индуктивным**.

Какое отношение все это имеет к компьютерам? Дело в том, что рекурсивный способ рассуждений реализован во многих языках программирования, в том числе - и в Паскале. Значит, этим языкам должен быть понятен и индуктивный способ написания программ.

Обозначим кратко факториал числа N, как $Factorial(N)$, и снова повторим наш индуктивный способ объяснения:

“Если N=1, то Factorial(N) = 1.
Если N>1, то Factorial(N) вычисляется умножением N на Factorial(N-1).”

В соответствии с этим объяснением напишем на Паскале функцию $Factorial$ для вычисления факториала:

```
FUNCTION Factorial(N: Byte): LongInt;
   BEGIN
      if N=1 then Factorial :=1;
      if N>1 then Factorial :=N* Factorial(N-1)
   END;
BEGIN
   WriteLn(Factorial(3))
END.
```

Обратите внимание, что в программе нигде не употребляется оператор цикла. Вся соль программы в том, что функция $Factorial$ вместо этого включает в себя вызов самой себя - $Factorial(N-1)$.

Что же происходит в компьютере во время выполнения программы? Механизм происходящего в точности соответствует нашему рассуждению по рекурсии.

Все начинается с того, что Паскаль пробует выполнить строку *WriteLn(Factorial(3))*. Для этого он вызывает функцию *Factorial*. Выполнение подпрограммы начинается с того, что в стеке отводится место для всех формальных параметров и локальных переменных, а значит и для нашего формального параметра *N*. Затем фактический параметр *3* подставляется на место формального параметра *N*, то есть в стек в ячейку *N* посылается *3*. Затем выполняется тело функции. Так как *3>1*, то Паскаль пытается выполнить умножение *3* Factorial(3-1)* и сталкивается с необходимостью знать значение функции *Factorial(2)*, для чего вызывает ее, то есть отправляется ее выполнять, недовыполнив *Factorial(3)*, но предварительно запомнив, куда возвращаться.

Спускаюсь на ступеньку ниже. В соседнем месте стека отводится место для *N*. Это уже другое *N*, путать их нельзя. В эту ячейку *N* посылается *2*. Затем выполняется тело функции. Пусть вас не смущает, что Паскаль второй раз выполняет тело функции, не закончив его выполнять в первый раз. Так как *2>1*, то Паскаль пытается выполнить умножение *2* Factorial(2-1)* и сталкивается с необходимостью знать значение функции *Factorial(1)*, для чего вызывает ее.

Спускаюсь еще на ступеньку. В соседнем месте стека отводится место еще для одного *N*. В эту ячейку *N* посылается *1*. Затем выполняется тело функции. Так как *1=1*, то Паскаль вычисляет *Factorial :=1*. Вот - впервые конкретное число. Затем Паскаль пытается выполнить следующую строку *if N>1 then Factorial :=N* Factorial(N-1)*. Поскольку нельзя сказать, что *1>1*, то выполнение тела функции закончено. Значит можно подниматься.

Поднимаюсь на одну ступеньку. Паскаль возвращается внутрь тела функции (той, где *N=2*) и успешно выполняет умножение - *Factorial :=2*1=2*.

Поднимаюсь еще на ступеньку. Паскаль возвращается внутрь тела функции (той, где *N=3*) и успешно выполняет умножение - *Factorial :=3*2=6*. Задача решена.

После выхода из подпрограммы место в стеке освобождается.

Итак, **рекурсией** в программировании называется вызов подпрограммы из тела самой подпрограммы.

Теперь поговорим о переполнении стека. Размер стека в Паскале не превышает 64K. В нашем случае в стеке одновременно хранилось три копии формальных параметров и локальных переменных. Если бы мы вычисляли факториал десяти, то копий было бы десять. В более сложных, чем факториал, задачах стек может легко переполниться, о чем Паскаль сообщает, когда уже поздно.

Чем хорош рекурсивный стиль программирования? В нашей программе о факториале мы как бы и не программировали вовсе, а просто обяснили компьютеру, что такое факториал. Как бы перешли на новый уровень общения с компьютером: вместо программирования - постановка задачи.

Чем плох рекурсивный стиль программирования? Если мы для решения той же задачи напишем программу не с рекурсией, а с обычным циклом, то такая программа будет выполняться быстрее и потребует меньше памяти.

Задание 124: Напишите рекурсивную функцию *fib* для вычисления чисел Фибоначчи.

13.7. Сортировка

Пусть имеется ряд чисел: *8 2 5 4*. Под **сортировкой** понимают их упорядочивание по возрастанию (*2 4 5 8*) или убыванию (*8 5 4 2*). Сортировать можно и символы (по алфавиту или коду ASCII) и строки (как слова в словаре).

Сортировка - очень распространенная вещь в самых разных программах, в частности - в системах управления базами данных.

Задача: Задан массив из 100 произвольных положительных чисел. Отсортировать его по возрастанию.

Если мы не можем сходу запрограммировать задачу, нужно подробно представить себе, в каком порядке мы решали бы ее вручную, без компьютера. Как бы мы сами сортировали 100 чисел? Мы бы запаслись пустым листом бумаги из 100 клеток. Затем нашли бы в исходном массиве максимальное число и записали его в самую правую клетку, а в исходном массиве на его месте записали бы число, меньшее самого маленького в массиве (в нашем случае подойдет 0). Затем нашли бы в изменившемся исходном массиве новое максимальное число и записали его на второе справа место. И так далее.

Вот программа для 10 чисел:

```
CONST  N              =  10;
TYPE   vector         =  array[1..N] of Word;
CONST  massiv_ishodn  :  vector =(3,8,4,7,20,2,30,5,6,9);  {Это исходный массив}
VAR    massiv_rezult  :  vector;                           {Это наш пустой лист бумаги}
       i              :  Word;
FUNCTION maximum (m:vector; N:Word; var Nomer_max: Word):Word; {Это вспомогательная функция для поиска
        максимума в массиве}
   VAR i,max:Word;
   BEGIN
      max:=m[1]; Nomer_max:=1;                              {Это порядковый номер максимального элемента }
      for i:=2 to N do if max<m[i] then begin max:=m[i]; Nomer_max:=i end;
      maximum:=max
   END;
PROCEDURE sortirovka (mass_ish:vector; N:Word; var mass_rez:vector);        {Основная процедура сортировки}
   VAR i, Nom_max:Word;
   BEGIN
      for i:=1 to N do begin
         mass_rez[N+1-i]:=maximum(mass_ish, N, Nom_max);
         mass_ish[Nom_max]:=0
      end{for};
   END;
BEGIN
   sortirovka (massiv_ishodn, N, massiv_rezult);
   for i:=1 to N do Write (massiv_rezult[i],' ');        {Распечатываем отсортированный массив}
END.
```

Функция *maximum*, кроме того, что сама имеет значение максимального элемента массива, выдает еще порядковый номер максимального элемента - *Nomer_max*. Это называется **побочным эффектом** функции.

Методов сортировки много. Приведенный метод - самый примитивный. Мало того, что нам пришлось расходовать память на второй массив, для выполнения сортировки массива из 100 элементов понадобилось бы около 100*100=10000 операций сравнения элементов массива между собой.

Существуют методы гораздо более эффективные. Приведу один из них - **метод пузырька**. Представьте себе тонкую вертикальную трубку с водой. Запустим снизу пузырек воздуха. Он поднимется до самого верха. Затем пустим еще один. Он поднимется наверх и остановится сразу же под первым. Затем запустим третий и так далее все сто пузырьков.

А теперь представим, что это не трубка, а наш исходный массив, а вместо пузырьков поднимаются максимальные элементы.

Вот алгоритм: Сравним первый элемент массива со вторым, и если второй больше, то ничего не делаем, а если первый больше, то меняем местами первый и второй элементы. В этом вся соль метода. Затем повторяем это со вторым и третьим элементами. Если третий больше, то ничего не делаем, а если второй больше, то меняем местами второй и третий элементы. Затем повторяем все это с третьим и четвертым элементами и так далее. Таким образом, максимальный элемент, как пузырек, поднимется у нас до самого верха.

Теперь, когда мы знаем, что элемент номер 100 у нас самый большой, нам предстоит решить задачу сортировки для массива из 99 элементов. Запускаем второй пузырек и так далее.

Метод пузырька не требует второго массива, да и сравнений здесь в два раза меньше.

Вот программа:

```
CONST  N       =  10;
TYPE   vector  =  array[1..N] of Word;
CONST  massiv  :  vector =(3,8,4,7,20,2,30,5,6,9);
VAR    i       :  Word;
PROCEDURE puziryok (var mass:vector; Razmer:Word);
   VAR i,m,c:Word;
   BEGIN
      for m:=Razmer downto 2 do begin      {Всего пузырьков – 9}
         for i:=1 to m-1 do begin          {i увеличивается – пузырек ползет вверх}
            if mass[i]>mass[i+1] then begin {Стоит ли обмениваться значениями}
               c:=mass[i];                {Три оператора для обмена значений двух элементов с помощью}
               mass[i]:= mass[i+1];       {транзитного элемента c}
               mass[i+1]:=c
            end{if}
         end{for};
      end{for};
   END;
BEGIN
   puziryok (massiv,N);
   for i:=1 to N do Write (massiv[i],' ');   {Распечатываем отсортированный массив}
END.
```

Задание 125: Отсортируйте по убыванию двумерный массив. Я имею в виду вот что:

$$a[1,1] >= a[1,2] >= \ldots >= a[1,N] >=$$
$$>= a[2,1] >= a[2,2] >= \ldots >= a[2,N] >=$$
$$>= a[3,1] >= a[3,2] >= \ldots$$

Глава 14. Строгости Паскаля

«Сердцем» этой главы является параграф «Синтаксические диаграммы», так как там материал о грамматике Паскаля представлен в наиболее строгом и упорядоченном виде. Так что после прочтения многих параграфов из этой главы стоит заглянуть в «Синтаксические диаграммы», чтобы окончательно разложить все по полочкам.

14.1. Структура программы

Самая маленькая программа на Паскале имеет такой вид:

BEGIN END.

Она, естественно, ничего не делает. Если мы хотим заставить программу что-то делать, то все операторы, приказывающие выполнять нужные нам действия, мы должны записать между *BEGIN* и *END*. Например:

BEGIN WriteLn(1993); WriteLn(1994) END.

Обычно программа содержит переменные, константы, обращения к подпрограммам и прочие элементы. Все они должны быть описаны выше *BEGIN*. Например:

```
CONST k = 10;
VAR   a : Real;
BEGIN
   a:=5;
   WriteLn(a+k)
END.
```

Таким образом, программа на Паскале состоит из двух и только двух разделов:

1) выше BEGIN расположен раздел описаний,
2) ниже BEGIN расположен раздел выполняемых операторов.

Выше этих двух разделов могут находиться две короткие строки, но о них чуть позже.

Приведем полный список служебных слов, после которых задаются описания:

- Переменные — описываются после служебного слова VAR
- Метки — описываются после служебного слова LABEL
- Константы — описываются после служебного слова CONST
- Процедуры — описываются после служебного слова PROCEDURE
- Функции — описываются после служебного слова FUNCTION
- Новые типы, определяемые программистом, описываются после служебного слова TYPE

Если программа на Паскале использует модули, то они должны быть перечислены выше раздела описаний после служебного слова *USES*.

И наконец, программа может иметь заголовок, который состоит из служебного слова **PROGRAM** и в простейшем случае имени программы.

Пример программы:

```
PROGRAM Divan;
USES    Crt,Graph;
LABEL   met1,met2;
CONST   k          =  100;
        S          =  'Хорошо!';
TYPE    Kniga      =  array [1..k] of String;
        Tablitsa   =  array [0..20,1..10] of Integer;
        Minuta     =  0..60;
VAR     x,y              :  Real;
        Uspevaemost      :  Tablitsa;
PROCEDURE Torpeda.....
FUNCTION  Invers.....
BEGIN
   .......
END.
```

14.2. Структура процедур и функций

Подпрограмма может не только обращаться к другим подпрограммам, но и иметь внутри себя свои собственные, внутренние, **вложенные подпрограммы**. Все эти подпрограммы описываются внутри данной подпрограммы, являются локальными в ней и не видны снаружи. Необходимость этого вытекает из желательности развязки отдельных частей больших проектов. Так, руководитель проекта может разделить проект на пять больших подпрограмм и поручить каждую отдельному руководителю. Те в свою очередь делят свои куски между подчиненными программистами, каждому поручая подпрограмму помельче, которая тоже в свою очередь нуждается в разбивке. В результате, если не использовать вложенности, проект оказывается разбит на несколько сотен маленьких равноправных подпрограмм. Нет никакой гарантии, что среди них не встретятся одноименные, и если бы вложенных подпрограмм не было, Паскаль обнаружил бы среди них одноименные и выдал ошибку. При использовании же вложенности это не страшно, так как подпрограммы с одинаковыми именами упрятаны внутрь более крупных подпрограмм и друг друга не видят.

Каждая из пяти подпрограмм большого проекта может иметь большой объем и руководитель этой подпрограммы должен обладать всем набором средств Паскаля. Поэтому каждая подпрограмма может иметь и свои внутренние типы, метки, переменные, константы, процедуры и функции. Все эти объекты описываются внутри данной подпрограммы, являются локальными в ней и не видны снаружи.

По сути структура подпрограммы копирует структуру программы за исключением того, что в подпрограммах запрещено писать *USES*.

Приведу пример записи программы с вложенными подпрограммами:

```
PROGRAM Hard;
USES   ...
LABEL  ...
CONST...
TYPE   ...
VAR    ...
PROCEDURE a1;
        CONST...
        VAR    ...
        PROCEDURE a11;
                LABEL  ...
                TYPE   ...
                VAR    ...
                BEGIN
                        ...
                END;
        FUNCTION f11
                VAR    ...
                BEGIN
                        ...
                END;
        BEGIN
                ...
        END;
FUNCTION  f2;
        PROCEDURE a21;
                BEGIN
                        ...
                END;
BEGIN
                ...
END;
BEGIN
        ...
END.
```

Здесь в программу *Hard* входят процедура *a1* и функция *f2*. В процедуру *a1* вложены процедура *a11* и функция *f11*. В функцию *f2* вложена процедура *a21*.

Из *f2* видна *a1*, но не видны *a11* и *f11*. Точно так же из *a21* видны *a1* и *f2*, но не видны *a11* и *f11*. Это значит, что в теле процедуры *a21*может содержаться вызов *a1*и *f2*, но не может содержаться вызов *a11*и *f11*.

14.3. Выражения

Понятие «выражение» я уже употреблял раньше без особых пояснений. Выражение – это то, что мы привыкли видеть в правой части оператора присваивания и в других местах. Например:

a := b+1	-	здесь выражение -	b+1
if c-6>f then …	-	здесь выражение -	c-6>f
WriteLn (a+5)	-	здесь выражение -	a+5

Прежде чем пояснить, что такое выражение вообще, приведу примеры наиболее распространенных типов выражений.

Арифметические выражения (то есть имеющие значением число):

- 0
- 2+5
- Sqrt(b+1) - Sqr(a[4,i]+r) + 1
- a[4,i] + vovka.ves
- ((w+b)/19)*(2/(w+1)+5)

Строковые выражения (то есть имеющие значением строку символов):

- ‘Весна’
- Copy(s,a,b)
- Copy(s,a,b)+ ‘Весна’

Логические выражения (то есть имеющие значением true или false):

- a>0
- (a+c)/(d+8)<=b+1
- c>’Ю’
- stroka=‘Весна’
- Copy(s,a,b)+ ‘Весна’ <> s1
- a in b

Вообще говоря, под **выражением** можно понимать произвольную имеющую смысл цепочку *операндов*, соединенных знаками операций (математических, логических и других) и круглыми скобками.

Под **операндом** будем понимать переменную простого типа, константу, элемент массива, поле записи, функцию и вообще любой объект, имеющий конкретное значение одного из простых типов.

Каждое выражение тоже обязано иметь конкретное значение одного из типов. Тип выражения определяется типами входящих в него операндов и операциями над этими операндами. Каким образом – об этом в следующем параграфе.

14.4. Совместимость типов

Часто при запуске программы Паскаль выдает сообщение *Type mismatch* (несовместимость типов) и на этом основании отказывается выполнять программу. Так произойдет, например, при выполнении ошибочного фрагмента *VAR y: Integer BEGIN y:=3/8: ….*, так как результат деления получается вещественный, а переменная *y* описана, как *Integer*. Нам нужно знать о том, какие типы совместимы, а какие нет.

Тип выражения с операндами разных типов. Что будет, если в одном выражении участвуют величины разных типов? Какого типа будет результат выражения? Например, какого типа будет сумма $a+b$ в следующем фрагменте:

```
VAR  a: Byte;  b: Word;
BEGIN  ....  a+b ...     ?
```

Вот ответ - Если осуществляется операция над двумя величинами разных типов, то перед выполнением операции они преобразуются к общему типу и только затем операция выполняется. Общий тип определяется так:

- Если диапазон одного из типов входит в диапазон другого, то общим типом становится тип с большим диапазоном. В приведенном выше примере значение переменной a будет преобразовано к типу *Word* и полученная сумма также будет иметь этот тип.
- Если диапазоны типов пересекаются, то общим типом становится ближайший тип, “охватывающий” оба типа.

  В примере

  ```
  VAR  a: Integer;  b: Word;
  BEGIN  ....  a+b ...
  ```

  значения *a* и *b* перед суммированием преобразуются к типу *LongInt*.
- Если один из типов вещественный, а другой целочисленный, то общий тип - всегда вещественный.

Когда в выражение входит несколько операндов (например, $a+b*c$) то указанные правила применяются последовательно, пока не будет вычислено все выражение (сначала определяется тип произведения $b*c$ и произведение вычисляется, затем исходя из получившегося типа определяется тип суммы и сумма вычисляется).

Требования к типам в операторе присваивания. Чтобы Паскаль мог выполнить оператор присваивания, тип переменной в левой части (*тип 1*) и тип выражения в правой части (*тип 2*) должны друг другу соответствовать по определенным правилам:

- эти типы должны быть одинаковы
- или, если эти типы не одинаковы, то диапазон типа 1 должен включать в себя диапазон типа 2, например:

Тип 1	*Тип 2*
(sun, mon, tue, wen)	(sun, mon, tue)
String	Char

- или *тип 1* - вещественный, *тип 2* – целочисленный

Требования к совместимости типов формальных и фактических параметров процедур и функций. Эти требования зависят от того, о каких параметрах идет речь – о параметрах-переменных или параметрах-значениях. В первом случае требование простое – типы должны быть эквивалентны. Во втором случае действуют требования к типам в операторе присваивания, где *тип 1* – формальный параметр, *тип 2* – фактический.

14.5. Форматы вывода данных

Обычный (бесформатный) вывод данных обладает тем недостатком, что данные, выводимые одним оператором WriteLn и перечисленные в скобках этого оператора через запятую, изображаются на экране подряд, без промежутков. Например, после выполнения фрагмента

c:='Ф'; s:='хорошо'; i:=18; WriteLn(c,s,i)

мы увидим в строке экрана

Ф	x	o	p	o	ш	o	1	8						

Мы можем приказать Паскалю отводить под каждое данное столько позиций в строке, сколько нам нужно:

WriteLn(c:3,s:8,i:4)

Вот что мы увидим в строке экрана:

		Ф			x	o	p	o	ш	o			1	8								

Здесь под *c* отведено поле в три позиции (с 1 по 3), начиная с левого края экрана. Под *s* отведено поле в 8 позиций (с 4 по 11). Под *i* отведено поле в 4 позиции (с 12 по 15). Информацией заполняется правая часть поля, а левая часть может остаться пустой.

Еще один недостаток бесформатного вывода: данные типа *Real* всегда изображаются в неудобочитаемом экспоненциальном виде. Например, после выполнения фрагмента

r:=465.28073; WriteLn(r)

мы увидим на экране

	4	.	6	5	2	8	0	7	3	0	0	0	0	2	3	1	E	+	0	0	0	2					

что означает 4.65280730000231 * 10^{2} или, что то же самое, 465.280730000231. Обратите внимание на откуда-то взявшиеся цифры 231. Их появление связано с неточностью представления вещественных чисел в компьютере.

Еще один пример: *r:=0.000000308; WriteLn(r)*

	3	.	0	8	0	0	0	0	0	0	0	0	0	7	9	E	-	0	0	0	7					

что означает 3.08000000000079 * 10^{-7} или, что то же самое, 0.000000308000000000079 .

Еще пример: *r:= -0.000003; WriteLn(r)*

-	2	.	9	9	9	9	9	9	9	9	9	9	9	5	3	E	-	0	0	0	6					

что означает -2.99999999999953 * 10^{-6} или, что то же самое, -0.0000029999999999953, а это практически равно -0.000003.

Как избавиться от экспоненциального вида? Формат *:9:3* прикажет Паскалю изобразить число типа Real в привычном для нас виде, отведя под него 9 позиций в строке, из них 3 позиции под дробную часть числа. Пример:

r:=465.28073; WriteLn(r:9:3)

		4	6	5	.	2	8	1															

Обратите внимание, что дробная часть округлена, так как она целиком не умещается в отведенный формат.

Еще пример: *r:=465.28073; WriteLn(r:10:0)*

							4	6	5														

Еще пример: *r:= -465.28073; WriteLn(r:10)*

-	4	.	7	E	+	0	0	0	2														

Это у нас получился вывод в "укороченном" экспоненциальном формате.

14.6. Переполнение ячеек памяти

После выполнения программы

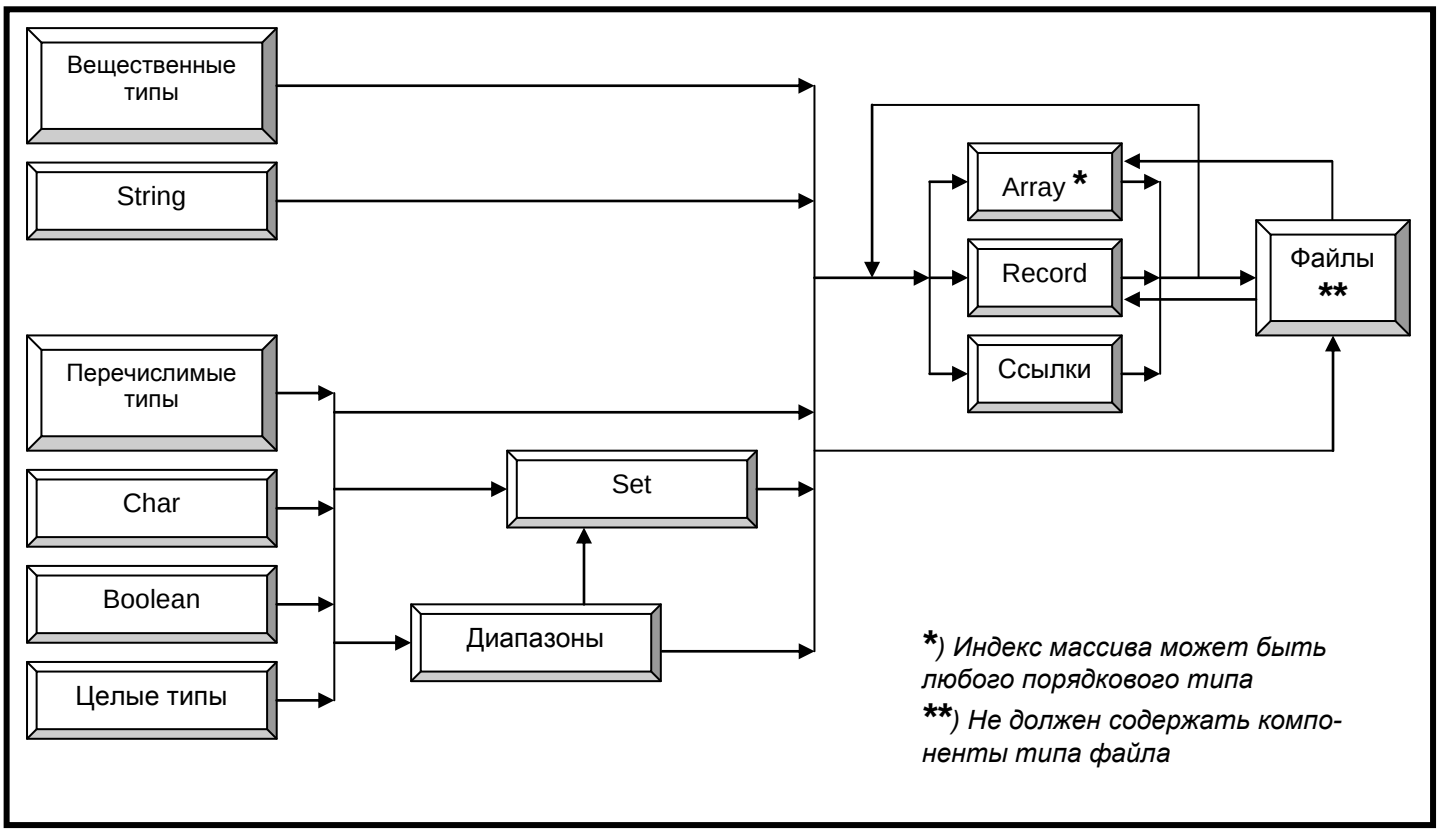
```
VAR i:Byte;
BEGIN   i:=250;  WriteLn(i);  i:=i+10;  WriteLn(i)   END.
```

мы увидим на экране числа 250 и 4. А почему не 250 и 260? Потому что тип Byte представляет только числа от 0 до 255. Если в процессе выполнения программы значение переменной вышло за диапазон своего типа, Паскаль не замечает ошибки, а в соответствующей ячейке оказывается ерунда. Это касается всех типов. Впрочем, ценой некоторой потери скорости Паскаль можно легко настроить на проверку выхода за диапазон (см. приложение).

14.7. Дерево типов

Паскаль предоставляет солидные возможности для конструирования новых типов. Так, если мы пишем *VAR a : array[1..10] of array [2..5] of set of Byte*, то имеем в виду, что переменная *a* является массивом *array[1..10]*, который сконструирован из массивов *array [2..5]*, каждый из которых сконструирован из элементов, являющихся множествами *set*, каждое из которых сконструировано из элементов типа *Byte*.

Схема, приводимая ниже, демонстрирует, какие типы из каких можно конструировать. Например, файлы можно конструировать из строк, а записи из массивов. А вот множества из вещественных чисел конструировать нельзя.

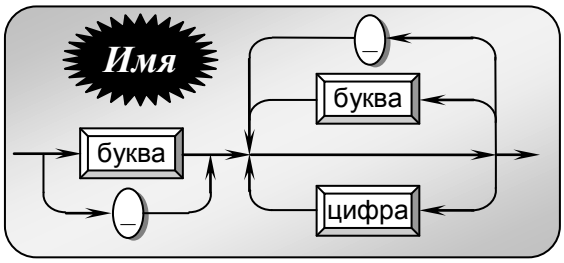


14.8. Синтаксисические диаграммы Паскаля

В этой книге я излагал Паскаль в основном на примерах, не стараясь очень уж строго давать правила записи операторов и других конструкций Паскаля. Тем не менее, когда Паскаль указывает вам на ошибку, вы, чтобы исправить ее, должны эти правила знать абсолютно точно.

Есть два самых распространенных способа записи синтаксиса. С одним из них я вас познакомил в 5.2, другой использует так называемые **синтаксические диаграммы**. Здесь я буду использовать второй метод плюс еще третий - обычный словесный (когда понятность будет достаточным возмещением за нестрогость).

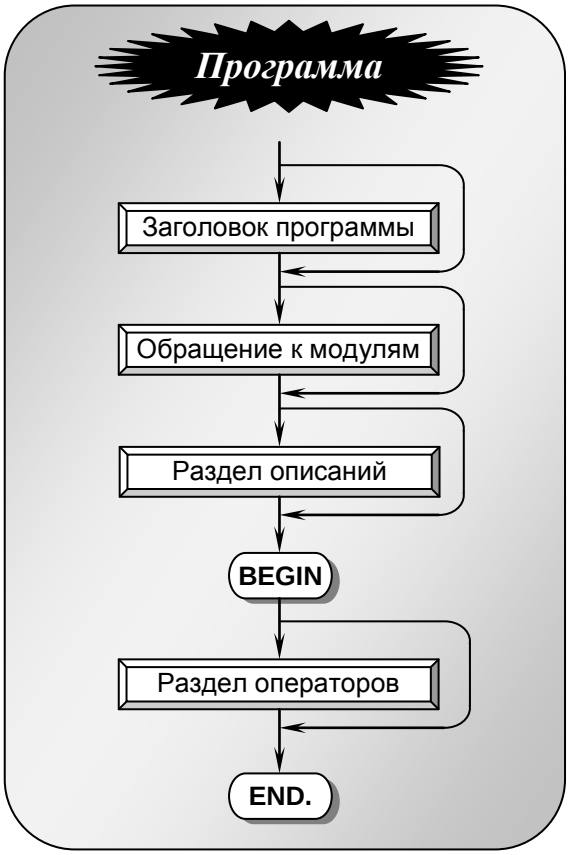
Синтаксическую диаграмму рассмотрим на примере. Все мы знаем, что имя - это цепочка букв, цифр и знаков подчеркивания, не начинающаяся с цифры. Это обычное словесное определение. А вот это же определение в виде синтаксической диаграммы:

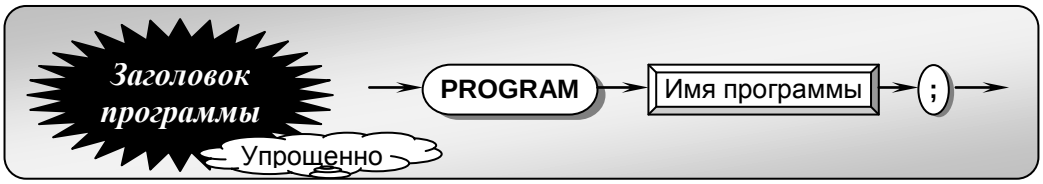


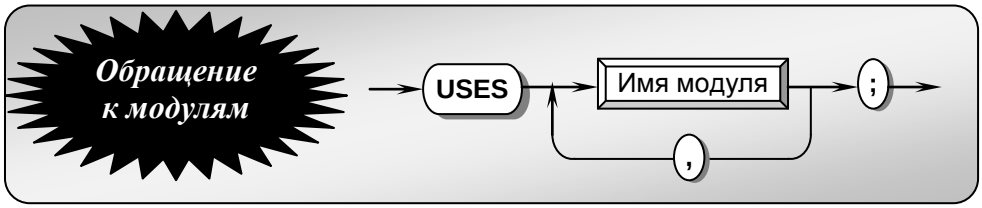
Пользоваться синтаксической диаграммой нужно так. Заходите в диаграмму по стрелке. Оказавшись внутри «домика», записывайте на лист бумаги в строчку то, что там указано. Предположим, в нашем случае вы зашли в «букву». Значит, записывайте любую букву, например, N. Затем продолжайте путешествие строго по стрелкам. Оказавшись на развилке, поворачивайте в любую сторону, разрешенную стрелками. Снова оказавшись в «домике», приписывайте справа в строчку то, что там указано. Пусть, например, следующий ваш домик «цифра». Вы должны приписать справа к букве любую цифру, например, 8. Получится $N8$. И так далее. Когда надоест, добирайтесь до выходной стрелки. Вы гарантированно получите правильное имя.

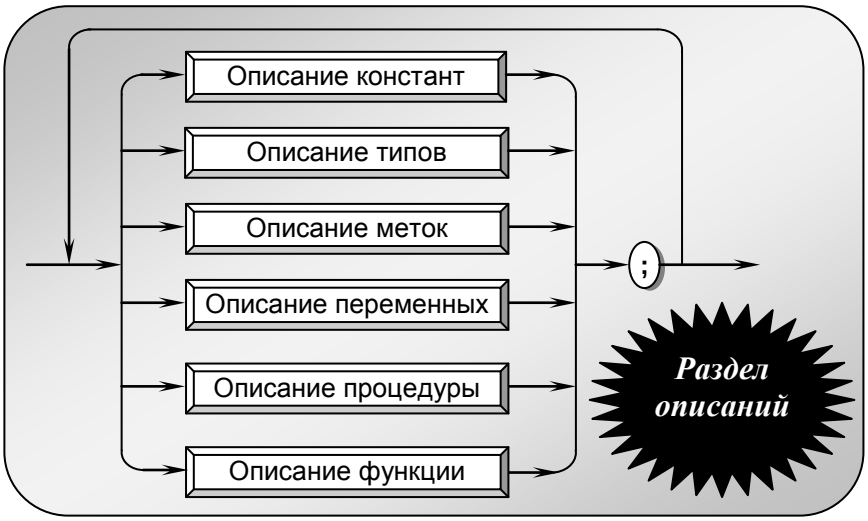
Сейчас я приведу синтаксические диаграммы всех изученных нами элементов Паскаля. Кое-какие элементы мы не проходили, кое-какие проходили упрощенно – соответствующие диаграммы тоже или будут отсутствовать, или будут нарисованы упрощенно, все такие случаи я буду оговаривать.

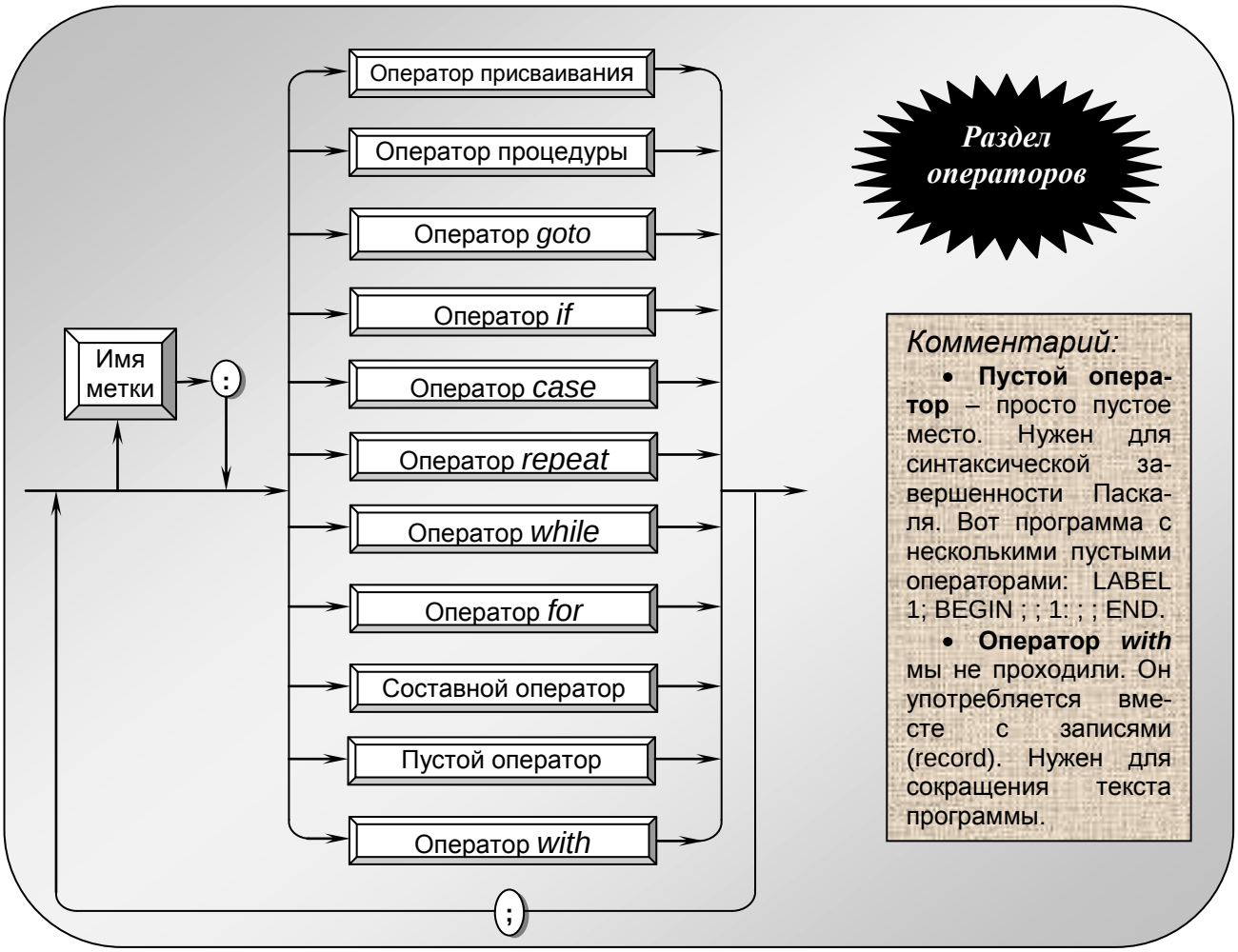
Также во избежание затемнения смысла подробностями диаграммы будут не везде до конца конкретны. Например, в диаграмме имени я писал “буква”, а надо было – “латинская буква”.





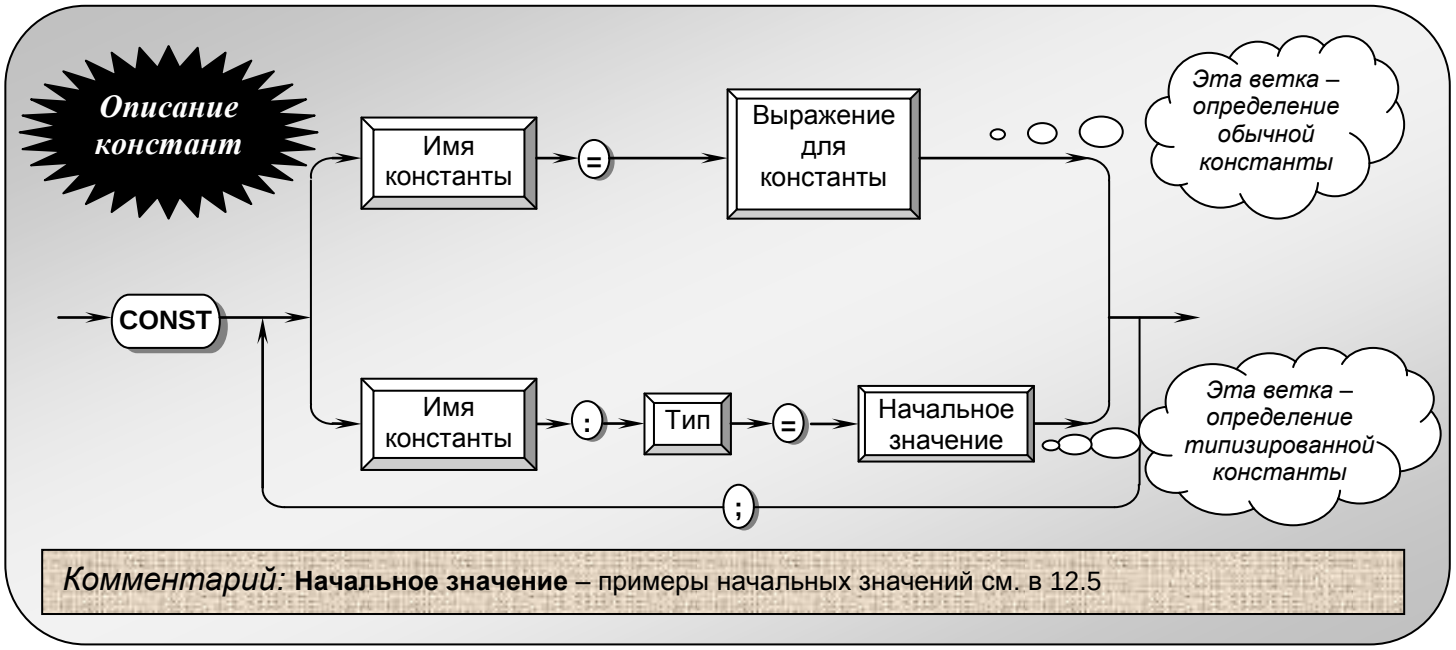


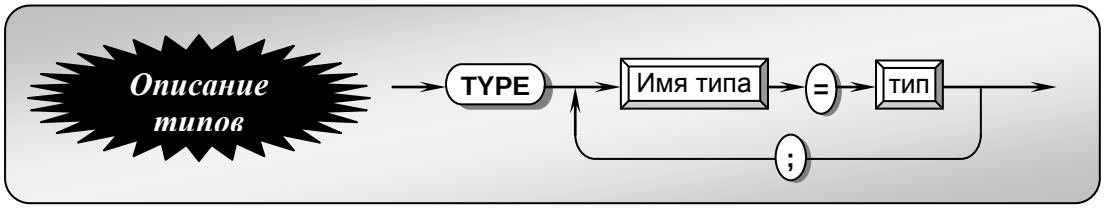


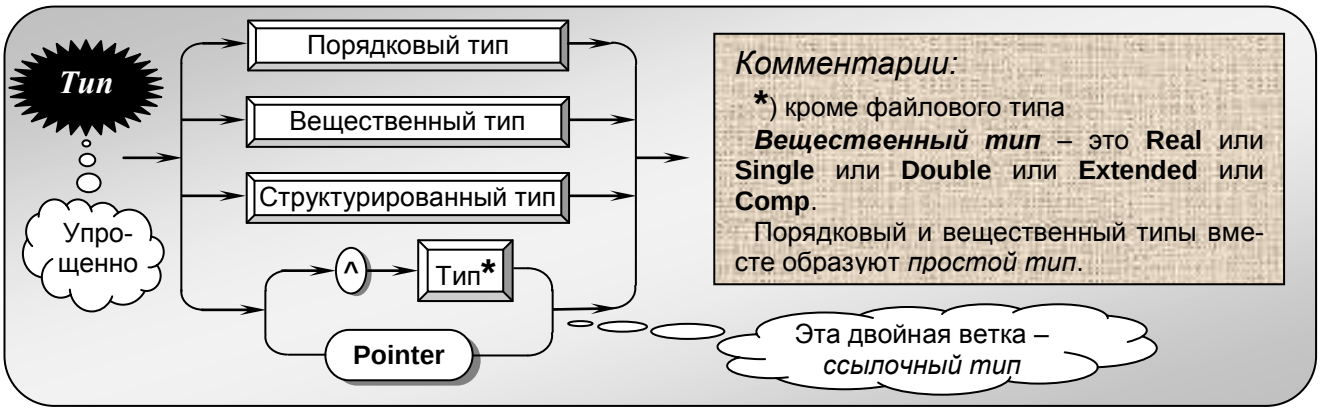


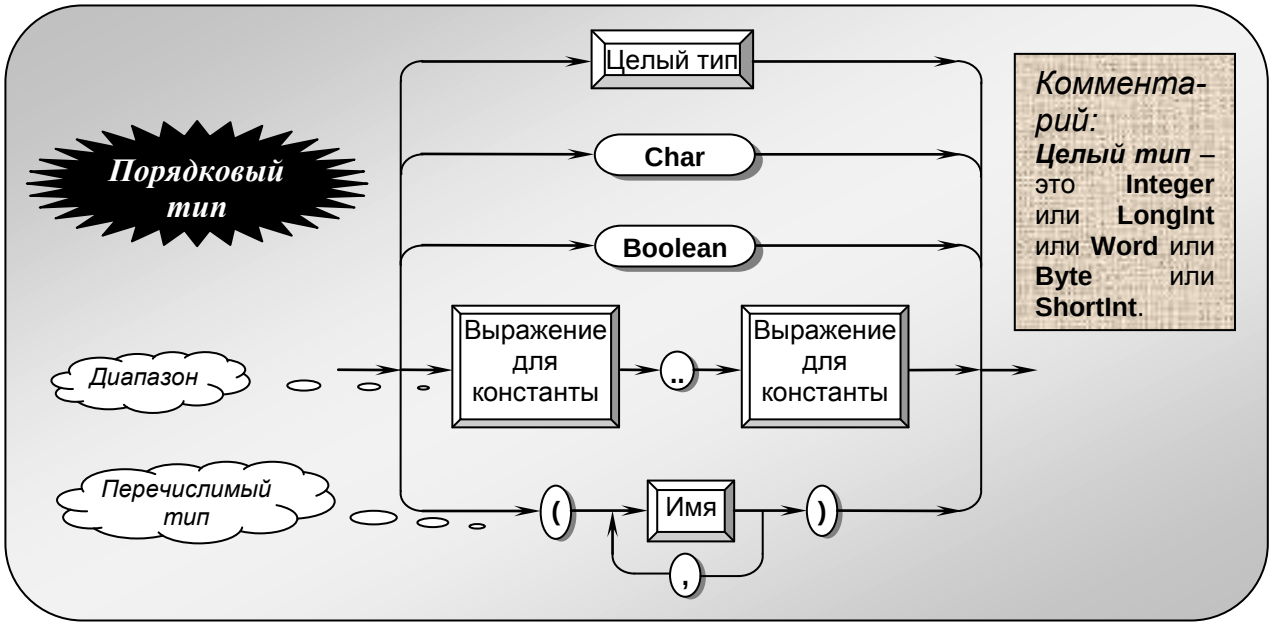
Комментарий:

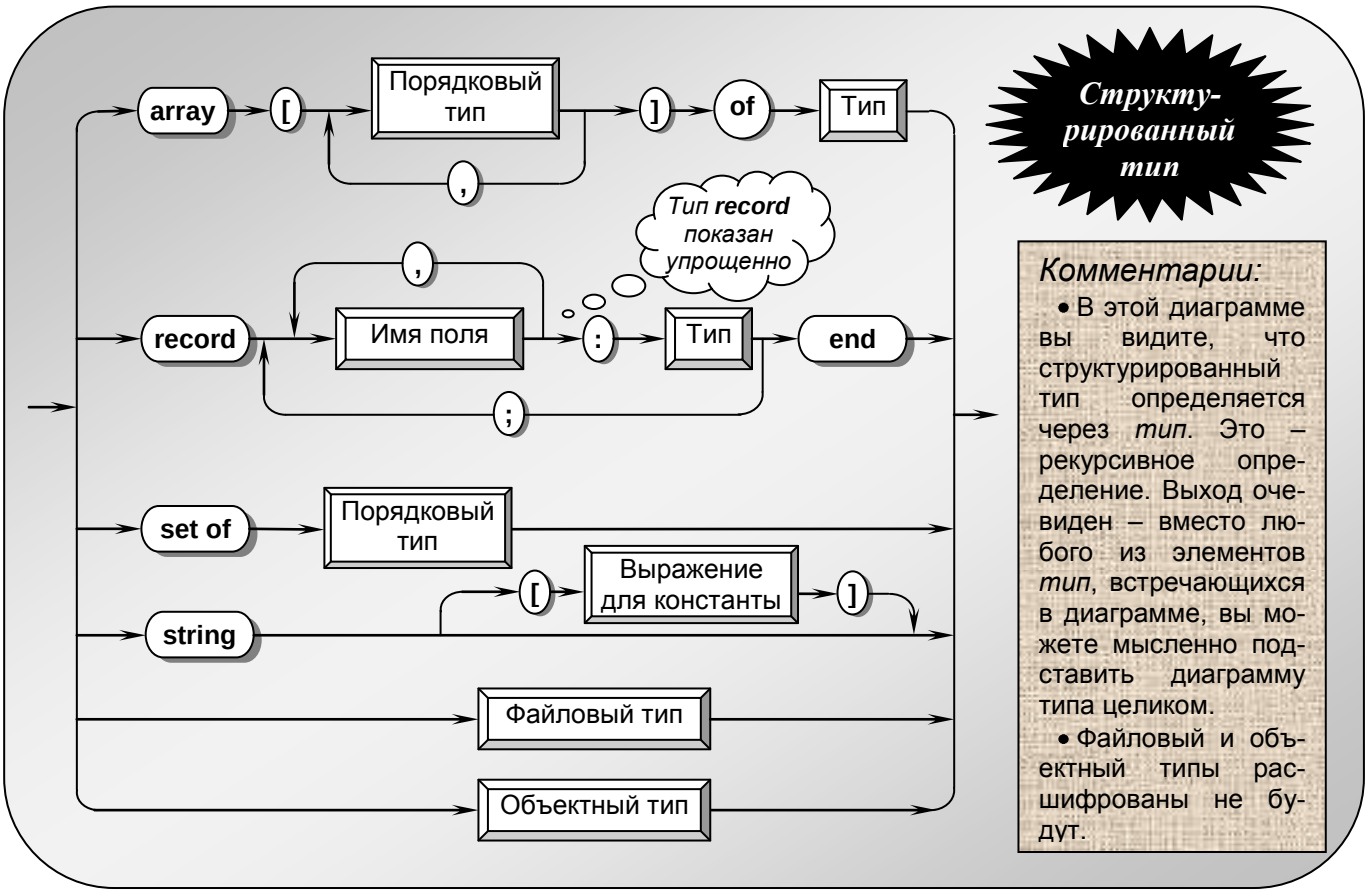
- **Пустой оператор** – просто пустое место. Нужен для синтаксической завершенности Паскаля. Вот программа с несколькими пустыми операторами: LABEL 1; BEGIN ; ; 1: ; ; END.
- **Оператор *with*** мы не проходили. Он употребляется вместе с записями (record). Нужен для сокращения текста программы.


Описание констант
Имя константы
=
Выражение для константы
Эта ветка – определение обычной константы
CONST
Имя константы
:
Тип
=
Начальное значение
Эта ветка – определение типизированной константы
;
Комментарий: Начальное значение – примеры начальных значений см. в 12.5


Описание типов
TYPE
Имя типа
=
тип
;

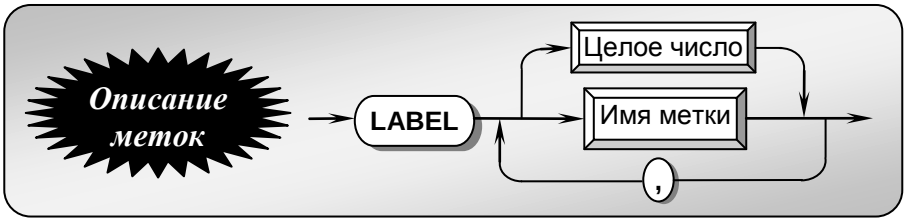

Тип
Порядковый тип
Вещественный тип
Структурированный тип
Упро-щенно
^
Тип*
Pointer
Комментарии:
*) кроме файлового типа
Вещественный тип – это Real или Single или Double или Extended или Comp.
Порядковый и вещественный типы вместе образуют простой тип.
Эта двойная ветка – ссылочный тип

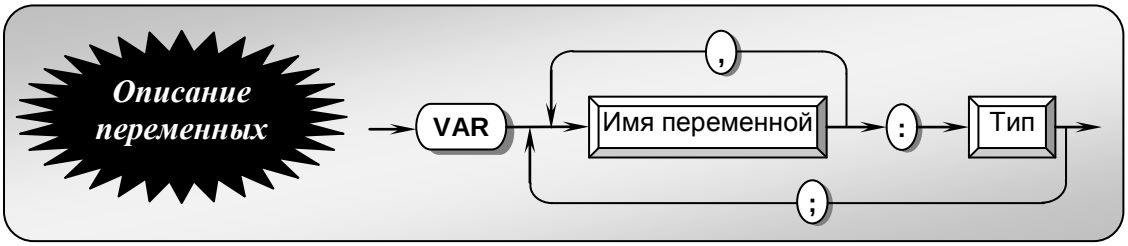

Порядковый тип
Целый тип
Char
Boolean
Диапазон
Выражение для константы
..
Выражение для константы
Перечислимый тип
(
Имя
)
,
Комментарий:
Целый тип – это Integer или LongInt или Word или Byte или ShortInt.

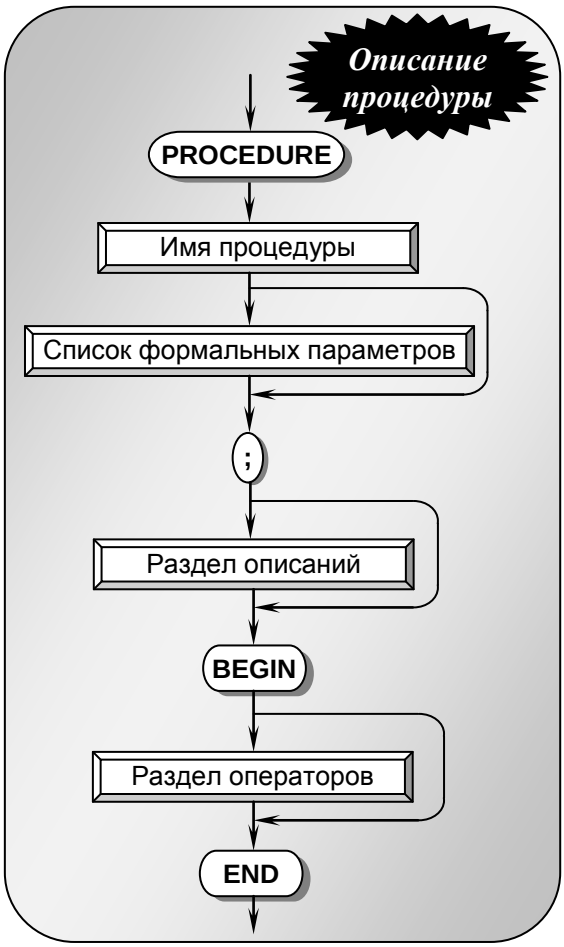


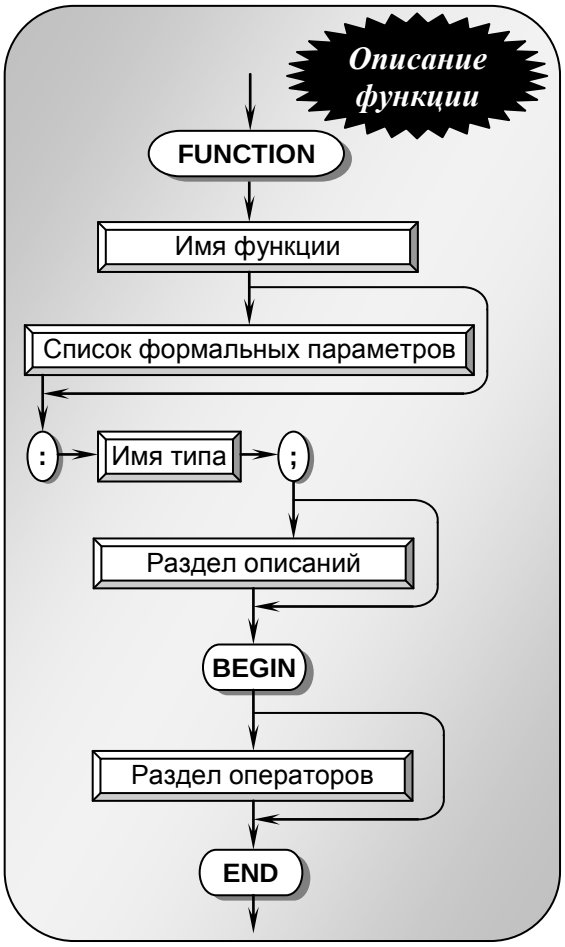
Комментарии:

- В этой диаграмме вы видите, что структурированный тип определяется через *тип*. Это – рекурсивное определение. Выход очевиден – вместо любого из элементов *тип*, встречающихся в диаграмме, вы можете мысленно подставить диаграмму типа целиком.
- Файловый и объектный типы расшифрованы не будут.

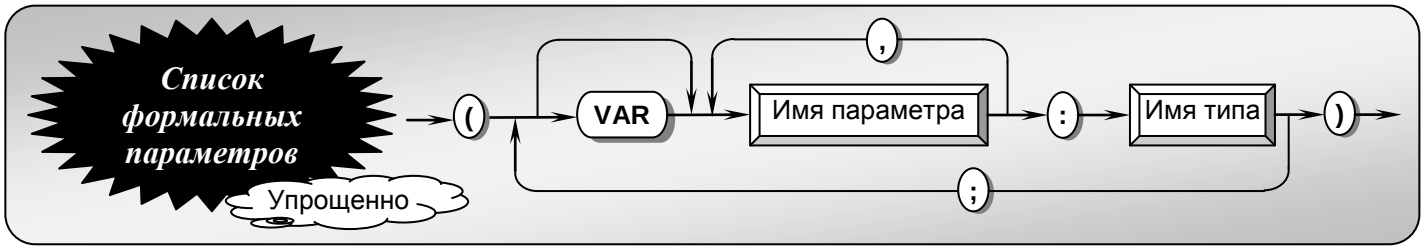


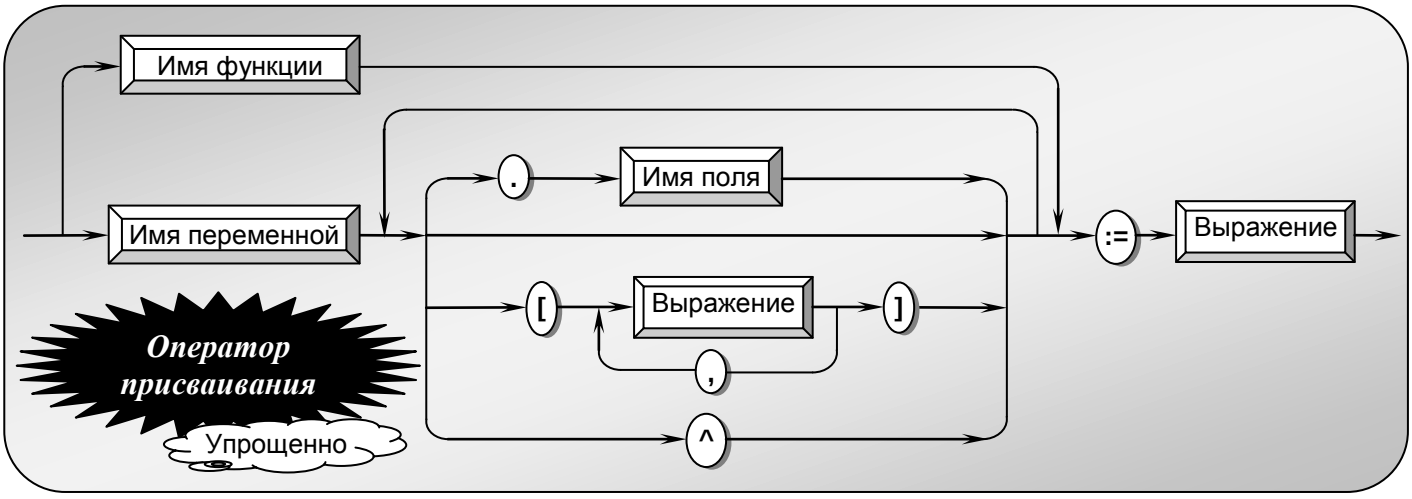


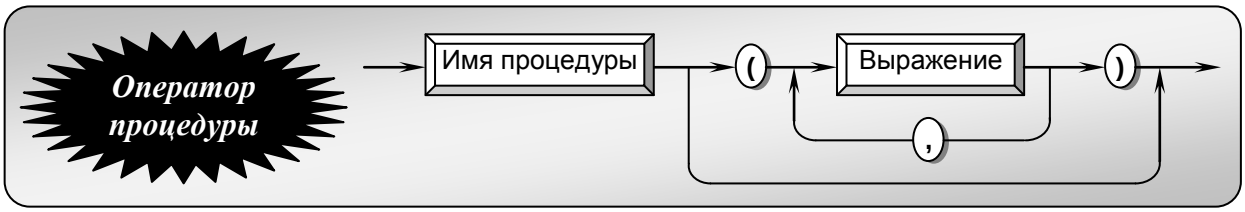


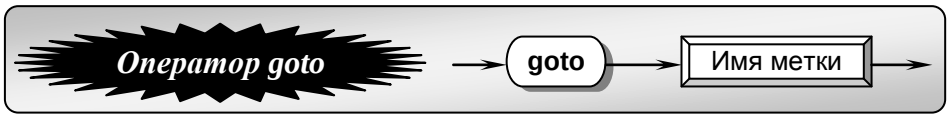


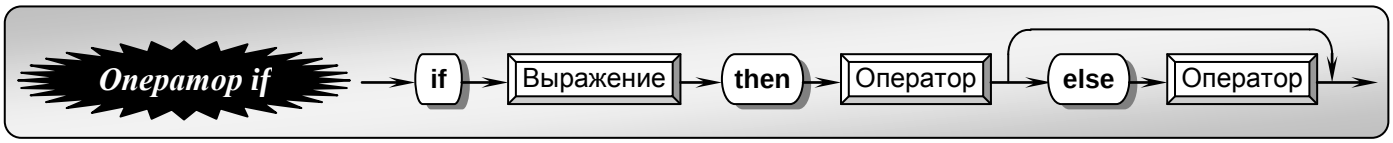
Опережающее описание процедуры или функции см. в 10.6

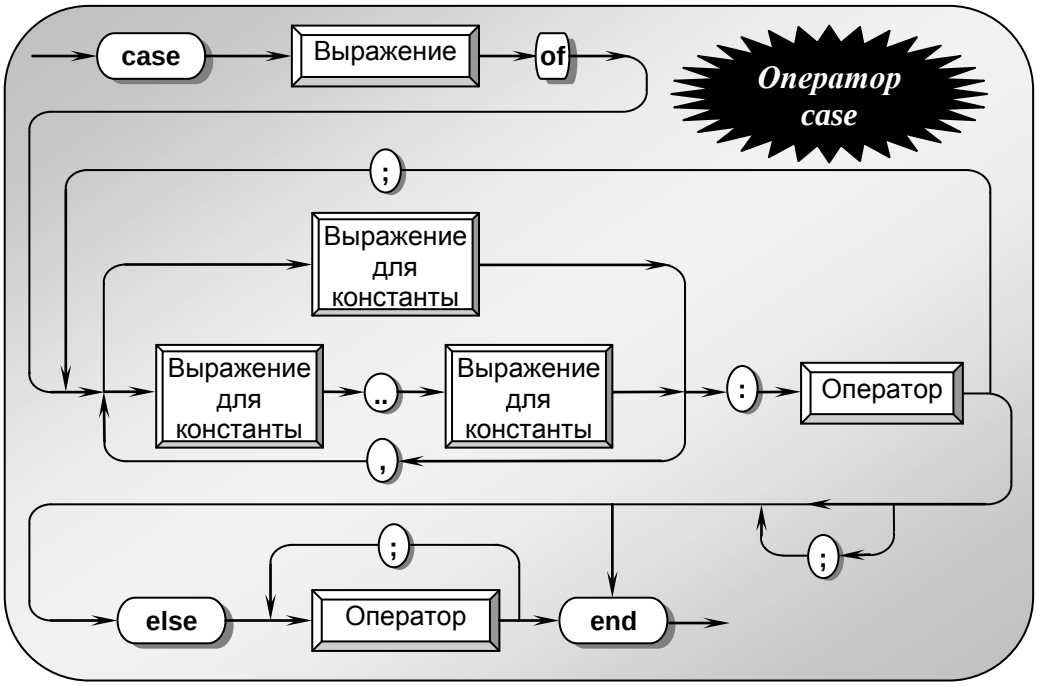


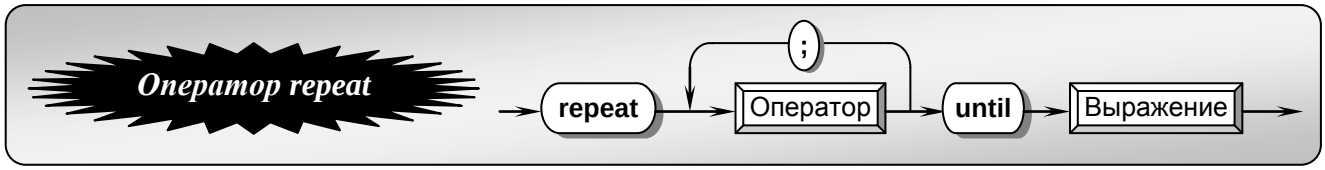


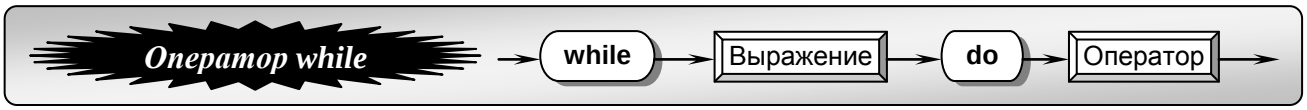

Оператор процедуры
Имя процедуры
(
Выражение
)
,

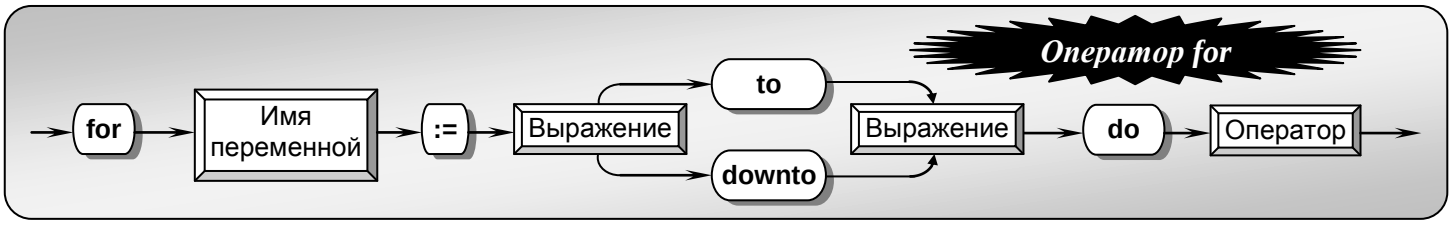

Оператор goto
goto
Имя метки

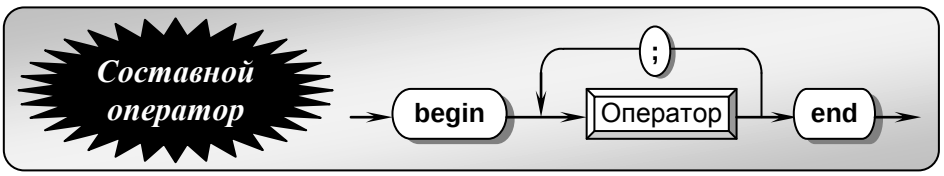

Оператор if
if
Выражение
then
Оператор
else
Оператор

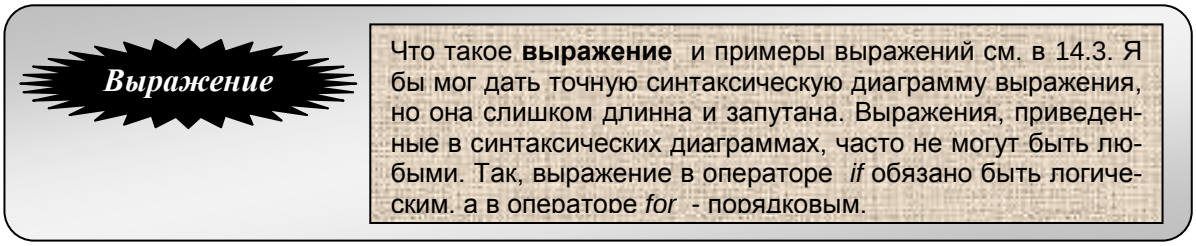

case
Выражение
of
Оператор case
;
Выражение для константы
Выражение для константы
..
Выражение для константы
:
Оператор
,
;
;
else
Оператор
end


Оператор repeat
repeat
;
Оператор
until
Выражение

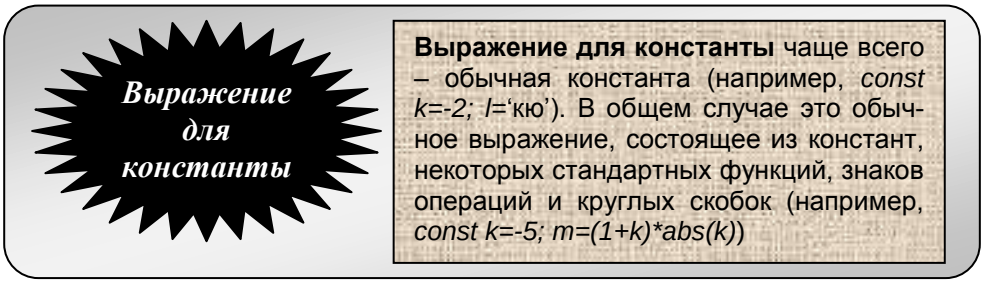




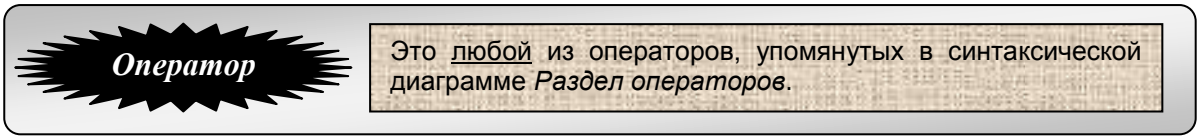




Что такое **выражение** и примеры выражений см. в 14.3. Я бы мог дать точную синтаксическую диаграмму выражения, но она слишком длинна и запутана. Выражения, приведенные в синтаксических диаграммах, часто не могут быть любыми. Так, выражение в операторе *if* обязано быть логическим, а в операторе *for* - порядковым.



Выражение для константы чаще всего – обычная константа (например, *const k=-2; l=*'кю'). В общем случае это обычное выражение, состоящее из констант, некоторых стандартных функций, знаков операций и круглых скобок (например, *const k=-5; m=(1+k)*abs(k)*)



Это любой из операторов, упомянутых в синтаксической диаграмме *Раздел операторов*.

Имя переменной, **имя константы** и все прочие имена, упомянутые в синтаксических диаграммах, образуются согласно синтаксической диаграмме *Имя*.

Глава 15. Другие возможности Паскаля

На этом со строгостями покончили. Нам осталось рассмотреть несколько дополнительных возможностей Паскаля.

15.1. Работа с файлами данных

Мы пока знакомы с выводом данных только на экран, а со вводом - только с клавиатуры. Сейчас мы познакомимся с выводом данных в файл и со вводом из файла. Если вы еще не знакомы с понятием файла или каталога, прочтите приложение. Для определенности мы будем считать, что файл расположен на магнитном диске, хотя файл - достаточно общее понятие, которое может применяться к различным устройствам ввода, вывода и хранения информации.

В Паскале существует три типа файлов. Мы познакомимся с самым распространенным из них - текстовым файлом. Вам совершенно не обязательно знать, как физически устроен носитель, на который файл будет записываться. При работе с текстовым файлом удобно представлять, что носитель - не диск, состоящий из дорожек, а подобен листу бумаги или экрану монитора, файл же - это строки информации на этом листе или экране. Данные в текстовых файлах могут быть целыми или вещественными числами, символами, строками.

Задача: Записать слово '*Азия*' и число *1998* на магнитный диск *c:* в текстовый файл с именем *fedos*, располагающийся в каталоге *PASCAL*.

Начнем с того, что придумаем файлу *fedos* псевдоним, которым мы будем пользоваться в программе. Пусть это будет *fepas*. Затем нам нужно объяснить Паскалю, что *fepas* - это текстовый файл, для чего мы начнем программу со строчки

*VAR fepas :**Text**;*

Раздел операторов начинаем с того, что объясняем Паскалю, какое настоящее имя соответствует псевдониму fepas:

***Assign**(fepas, 'c:\PASCAL\fedos');*

Затем переместим магнитную головку в начало файла для записи информации в файл (**откроем** файл для записи):

***Rewrite**(fepas);*

Теперь запишем в файл нужную информацию:

***WriteLn**(fepas, 'Азия');*

WriteLn(fepas, 1998);

Поскольку мы не собираемся ничего больше писать в файл, то закрываем его для записи:

***Close**(fepas);*

Вот программа целиком:

```
VAR fepas  :Text;
BEGIN
   Assign(fepas,   'c:\PASCAL\fedos');
   Rewrite(fepas);
   WriteLn(fepas, 'Азия');
   WriteLn(fepas, 1998);
   Close(fepas);
END.
```

После выполнения программы вы обнаружите в файле fedos две строки:

Азия
1998

Если бы вы вместо *WriteLn* использовали *Write*, то строка была бы одна:

Азия1998

Если к моменту выполнения программы файл *fedos* не существовал, то процедура *Rewrite* создаст пустой файл с таким именем в указанном каталоге. Если существовал, то стирает его содержимое.

Если вы не хотите стирать содержимое файла, а просто хотите дописать что-нибудь в его конец, то процедура *Rewrite* вам не подойдет. Вам вместо нее нужно будет употребить процедуру ***Append***. Вот ее вызов - *Append(fepas)*.

Если вы знаете, какой каталог во время выполнения программы является *текущим*, и хотите, чтобы файл *fedos* был создан в текущем каталоге, вы можете записать просто

Assign(fepas, 'fedos')

Информация в наш файл может записываться только по порядку, последовательно. Мы не можем записать что-то сперва в начало файла, потом в конец, потом в середину. То же самое относится и к считыванию, о котором сейчас пойдет речь.

А теперь решим обратную задачу: В некоем файле *fedos* записаны строки. Вывести первую и третью из них на экран монитора.

```
VAR  fepas :Text;
     a,b,c  :String;       {Три переменные в оперативной памяти, в которые будут
                            записаны первые три строки из файла}
BEGIN
   Assign(fepas,   'c:\PASCAL\fedos');
   Reset(fepas);           {Переместим магнитную головку в начало файла для считывания
                            информации из файла (откроем файл для считывания) }
   ReadLn (fepas, a);      {Читаем первую строку из файла}
   ReadLn (fepas, b);      {Читаем вторую строку из файла}
   ReadLn (fepas, c);      {Читаем третью строку из файла}
   Close(fepas);
   WriteLn(a);             {Печатаем первую строку из файла}
   WriteLn(c);             {Печатаем третью строку из файла}
END.
```

Как видите, для того, чтобы добраться до третьей строки, нам пришлось прочитать вторую.

Если третья строка числовая, то можно было бы описать переменную *c*, как числовую, а не строковую.

А теперь напишем программу, которая в цикле записывает в файл 10 строк, а затем для проверки их считывает.

```
VAR  f      :Text;
     a,b    :String;
     i      :Byte;
BEGIN
   Assign(f,'c:\PASCAL\textik.txt');     {Обозначим файл textik.txt из каталога PASCAL именем f}
   Rewrite(f);                 {Переместим магнитную головку в начало файла для
                                записи информации в файл (откроем файл)}
   WriteLn('Введите с клавиатуры 10 произвольных строк');
   for i:=1 to 10 do begin
      ReadLn(a);               {Ввести с клавиатуры в оперативную память произвольную строку текста}
      WriteLn(f,a)             {Записать эту строку из оперативной памяти в файл}
   end;
   Close(f);                   {Закрываем файл для записи}

   WriteLn('А теперь посмотрим, что мы записали в файл:');
   Reset(f);                   {Переместим магнитную головку в начало файла для
                                считывания информации из файла (откроем файл)}
   for i:=1 to 10 do begin
      ReadLn(f,b);             {Переслать из файла в оперативную память строку текста}
      WriteLn(b)               {Послать эту строку из оперативн.памяти на монитор}
   end;
   Close(f);                   {Закрываем файл для чтения}
END.
```

Если вы хотите прочесть текстовый файл, но не знаете, сколько там строк, то вам нужно какое-то средство, чтобы компьютер определил, когда заканчивается файл. Это средство - функция ***EOF*** (сокращение от *end of file – «конец файла»*). Вот фрагмент, решающий дело:

while NOT EOF(fepas) do ... ReadLn(fepas, ...)
пока нет конца файла fepas делай ...

Задание 126: “База данных ваших школьных оценок”. Вы завели файл, в который записываете свои школьные оценки. Каждый раз, получив оценку, вы дописываете в файл оценку и предмет, по которому оценка получена. Создайте 4 программы:

1) Для дозаписи в конец файла очередной оценки и предмета.
2) Для вывода на экран всего файла.
3) Для вывода на экран всех отметок по заданному предмету. (Для определенности договоримся, что больше 1000 оценок в год вы не получите.)
4) Для определения, сколько было по заданному предмету таких-то оценок (скажем, троек).

Оформите каждую из четырех программ, как процедуру, и создайте единую программу - "Систему управления базой данных", которая начинает работу с вопроса*: "Чем будем заниматься - дозаписывать, выводить весь файл ...?*" и в зависимости от ответа запускает одну из четырех процедур.

Задание 127: Вы можете помочь адмиралу из 12.13 и организовать чтение из файла всей нужной информации о подлодках.

Задание 128: Многие компьютерные игры позволяют "сохраняться", то есть в любой момент игры при нажатии определенной клавиши записывать в файл все данные об этом моменте, чтобы в следующий раз, когда вы сядете играть, начинать не сначала, а с того момента, в который вы записались. В игре "Торпедная атака" организуйте при нажатии на клавишу *S* сохранение, то есть запись в файл имени игрока, уровня игры, количества выстрелов и количества попаданий. А при запуске игра должна спрашивать, будем ли возобновлять сохраненную игру или начинать новую.

15.2. Вставка в программу фрагментов из других программных файлов

Теперь рассмотрим совсем другие файлы – те, в которых вы храните свою программу. Предположим, вы с другом решили создать программу из 7 процедур. Вы делаете 3 процедуры и тело программы, а ваш друг - 4 процедуры. Друг записал все 4 процедуры в файл *VSTAVKA.pas* на вашем компьютере. Вот он:

```
PROCEDURE fa ; BEGIN Sound(698); Delay(300); NoSound  END;
PROCEDURE sol; BEGIN Sound(784); Delay(300); NoSound  END;
PROCEDURE la ; BEGIN Sound(880); Delay(300); NoSound  END;
PROCEDURE si ; BEGIN Sound(988); Delay(300); NoSound  END;
```

Конечно, файл вашего друга не является законченной программой и сам по себе не запускается.
А вот ваша программа:

```
USES CRT;
PROCEDURE doo;    BEGIN Sound(523); Delay(300); NoSound  END;
PROCEDURE re ;    BEGIN Sound(587); Delay(300); NoSound  END;
PROCEDURE mi ;    BEGIN Sound(659); Delay(300); NoSound  END;
BEGIN
   doo;re;mi;fa;sol;la;si;la;sol;fa;mi;re;doo
END.
```

Теперь вам нужно собрать оба куска в единую программу. Для этого вы можете скопировать текст из файла вашего друга в свой файл (как это делается, рассказано в части IV). Но если вы не хотите этого делать, чтобы, скажем, не увеличивать свой файл, вы можете воспользоваться директивой компилятора *$I*. **Директива компилятора** - это специальная инструкция, вставленная в текст вашей программы на Паскале и предназначенная для управления компьютером на этапе компиляции вашей программы. Директива компиляции имеет вид ***{$....}*** и Паскаль не путает ее с обычным комментарием только из-за наличия значка доллара. Символы, стоящие после значка доллара, и являются управляющей информацией для компилятора. Директива ***{$I*** *c:\PASC\F25}* является приказом компилятору подставить в это место текст, находящийся в файле *F25* из каталога *PASC* диска *c*. Если файл находится в текущем каталоге, то достаточно указать его имя. Вот ваша готовая к работе программа с директивой:

```
USES CRT;
PROCEDURE doo;    BEGIN Sound(523); Delay(300); NoSound  END;
PROCEDURE re ;    BEGIN Sound(587); Delay(300); NoSound  END;
PROCEDURE mi ;    BEGIN Sound(659); Delay(300); NoSound  END;

   {$I VSTAVKA}  {Директива компилятору на вставку текста из файла VSTAVKA}

BEGIN
   doo;re;mi;fa;sol;la;si;la;sol;fa;mi;re;doo
```

```
END.
```

15.3. Модули программиста

Известно, что в Паскале нет стандартной функции, возводящей число в целую неотрицательную степень. В целую степень, большую двух. Предположим, что вы математик, и вам нужна функция, возводящая число в любую целую неотрицательную степень. Напишем эту функцию:

```
FUNCTION st(a:Real;  n:Word) :Real;
   VAR  step   :Real;
        i      :Word;
   BEGIN
      step:=1;
      for i:=1 to n do step:=step*a;
      st:=step
   END;
BEGIN
   WriteLn(st(2,3))     {Это 2 в кубе, то есть 8}
END.
```

Пусть вы часто пишете программы, использующие возведение в степень. Но вам лень в каждую такую программу вставлять описание функции *st*. Вы можете пойти двумя путями:

- Описать *st* и другие часто встречающиеся процедуры и функции в другом файле и использовать директиву *$I*.
- Описать *st* и другие часто встречающиеся процедуры и функции в другом файле и оформить этот файл, как новый **модуль**.

Второй способ немного сложнее, но намного лучше первого. Вот как будет выглядеть ваш модуль:

```
UNIT Mathemat;          {Заголовок модуля с придуманным вами именем}

INTERFACE               {Раздел ИНТЕРФЕЙСА}
FUNCTION st(a:Real; n:Word) :Real;

IMPLEMENTATION   {Раздел РЕАЛИЗАЦИИ}
FUNCTION st;
      VAR  step   :Real;
           i      :Word;
      BEGIN
         step:=1;
         for i:=1 to n do step:=step*a;
         st:=step
      END;

BEGIN                   {Раздел ИНИЦИАЛИЗАЦИИ, у нас он пуст}
END.
```

Вам нужно просто ввести этот текст, как обычную программу, в новое окно текстового редактора и сохранить на диске под именем *Mathemat.pas*, так как имя файла, в котором расположен модуль, должно совпадать с именем модуля. Однако, модуль не является программой и не может быть запущен на выполнение сам по себе. Пользоваться вашим новым модулем вы можете так же, как обычным стандартным. Вот ваша программа, вызывающая модуль:

```
USES Mathemat;
BEGIN
   WriteLn(st(10,6)  :20:4);        {Это 10 в шестой степени}
   WriteLn(st(5, 3)  :20:4);        {Это 5 в кубе}
END.
```

Файл-модуль на первых порах сохраняйте в том же каталоге, что и файл вызывающей его программы.

В разделе интерфейса вы приводите заголовки предназначенных для использования процедур, функций, а также описания типов, переменных и констант, которые вы также не хотите каждый раз описывать в вызывающей программе. В общем, все то, чем могут пользоваться другие программы.

В разделе реализации вы приводите краткие заголовки процедур и функций, полные заголовки которых приведены в разделе интерфейса, и описываете их тела. Если этим процедурам и функциям понадобятся для работы вспомогательные процедуры и функции, типы, константы и переменные, то они тоже описываются в разделе реализации.

Когда вы в первый раз запустите на выполнение программу, вызывающую ваш новый модуль, этот модуль откомпилируется и сохранится на диске под именем *Mathemat.tpu*. В следующий раз будет использоваться именно он, а не *Mathemat.pas* (до тех пор, пока вы не измените текст модуля).

Рассмотрим еше один пример. Предположим, что вы часто пишете графические программы и вам надоело в каждой программе инициализировать графику. К тому же вы недовольны, что стандартный модуль Graph позволяет вам рисовать кружочки и квадратики, но не позволяет рисовать крестики и треугольники. И наконец, вы бы хотели, чтобы в начале работы любой вашей программы экран был бы обведен золотой рамочкой. Вот модуль, решающий эти задачи:

```
UNIT Mygraph;

INTERFACE                       {Раздел ИНТЕРФЕЙСА}
PROCEDURE krest(x_tsentr, y_tsentr, razmer:Word);
                                {Задаются координаты центра и размер креста}
PROCEDURE treug(x1, y1, x2, y2, x3, y3 :Word);
                                {Задаются координаты трех вершин треугольника}

IMPLEMENTATION                  {Раздел РЕАЛИЗАЦИИ}
USES Graph;                     {Без этого не будет работать процедура Line}
PROCEDURE krest;  BEGIN
      Line(x_tsentr-razmer, y_tsentr, x_tsentr+razmer, y_tsentr);
      Line(x_tsentr, y_tsentr-razmer, x_tsentr, y_tsentr+razmer);
                  END;
PROCEDURE treug;  BEGIN
      Line(x1,y1,x2,y2);
      Line(x2,y2,x3,y3);
      Line(x3,y3,x1,y1);
                  END;
                                {Раздел ИНИЦИАЛИЗАЦИИ}
VAR d,m :Integer;               {Переменные для инициализации графики}
BEGIN
   d:=0;
   InitGraph(d,m,'<путь к гр.др>');{Инициализация графики}
   SetColor(Yellow);            {Рисуем рамочку}
   SetLineStyle(0,0,ThickWidth);
   Rectangle(10,10,630,470);
   SetColor(White);             {Возвращаем нормальный цвет}
   SetLineStyle(0,0,NormWidth)  {Возвращаем нормальную толщину линии}
END.
```

Если вы хотите, чтобы при запуске программы, использующей модуль, каждый раз перед выполнением самой программы автоматически выполнялись какие-то действия, вы задаете соответствующие операторы в разделе инициализации. Если для этого нужны константы, типы и переменные, они описываются в разделе реализации.

Вот программа, чертящая крест, треугольник и кружок:

```
USES Mygraph,Graph;
BEGIN
   treug(500,50,600,300,450,450);
   krest(200,150,80);
   Circle(100,350,40);
   ReadLn;
END.
```

Обращение здесь к модулю Graph понадобилось только из-за желания нарисовать кружок.

Использование модулей лучше использования директивы *$I* хотя бы по двум причинам:

- Модуль уже откомпилирован и не требует каждый раз компиляции.
- Объем программы без модулей не может превышать 64К. Каждый модуль может вмещать в себя дополнительные 64К.

Задание 129: Если хотите, создайте себе модули *Music*, *Graphica* или какие-нибудь другие.

15.4. Дополнительные процедуры и функции модуля Graph

Кроме процедур, которые заставляют Паскаль что-либо сделать, в модуле Graph имеются функции, которые могут сообщать программисту ту или иную информацию. Вот некоторые из них:

Функция	Смысл
GetMaxX :Integer	Выдает максимально возможную горизонтальную координату экрана
GetMaxY :Integer	Выдает максимально возможную вертикальную координату экрана
GetPixel(x,y :Integer) :Word	Выдает номер цвета пиксела с координатами *x* и *y*.

А вот еще процедуры рисования:

Процедура	Смысл
Arc (x,y :Integer; fi1,fi2,r :Word)	Рисует дугу окружности с центром в точке *x,y* и радиусом *r*. Дуга начинается от угла *fi1* градусов и кончается углом *fi2* градусов.
PieSlice (x,y :Integer; fi1,fi2,r :Word)	Закрашенный сектор круга. Дуга сектора определяется так же, как в процедуре *Arc*. Цвет и стиль заливки определяются процедурой *SetFillStyle*.
FillEllipse (x,y :Integer; rx,ry :Word)	Закрашенный эллипс с центром в точке *x,y* и радиусами *rx,ry*. Цвет и стиль заливки определяются процедурой *SetFillStyle*.
Sector (x,y :Integer; fi1,fi2,rx,ry :Word)	Закрашенный сектор эллипса. Опирается на дугу эллипса с центром в точке *x,y* и радиусами *rx,ry*. Дуга начинается от угла *fi1* градусов и кончается углом *fi2* градусов.
Bar (x1,y1, x2,y2:Integer)	Закрашенный прямоугольник с противоположными углами в точках *(x1,y1)* и *(x2,y2)*
Bar3D (x1,y1, x2,y2:Integer; tol:Word; top:Boolean)	Трехмерный параллелепипед, обращенный к нам прямоугольной гранью с противоположными углами в точках *(x1,y1)* и *(x2,y2)*. Толщина параллелепипеда – *tol*. Если *top* равно *TopOff*, то параллелепипед – без верха, если *top* равно *TopOn*, то – с верхом.

15.5. Копирование и движение областей экрана

До сих пор мы заставляли двигаться лишь простые объекты: окружности, квадраты, линии. Если же мы хотим заставить двигаться что-нибудь посложнее, например, снеговика из 9.3, то нам придется изрядно потрудиться. Чтобы не рисовать и не стирать по-очереди все элементы, из которых состоит снеговик, мы можем использовать процедуры *GetImage* и *PutImage*, которые позволяют копировать любую прямоугольную область экрана целиком в другое место экрана.

Пример 1. Нарисуем в левом верхнем углу экрана четыре окружности, а затем скопируем получившиеся «очки» в правый нижний угол.

```
USES    Graph;
VAR     Gd, Gm   : Integer;
        P        : pointer;
        Size     : Word;
BEGIN
   Gd := 0; InitGraph(Gd, Gm, 'c:\tp\bgi');
   SetLineStyle(0,0,Thickwidth);
        {Рисуем очки:}
   Circle(50,100,20);    Circle(50,100,15);
   Circle(90,100,20);    Circle(90,100,15);
        {В целях наглядности нарисуем также диагональ экрана:}
   Line(0,0,640,480);
```

```
    Size := ImageSize(10,60,120,140);
    GetMem(P, Size);
    GetImage(10,60,120,140,P^);
    ReadLn;
    PutImage(500,400, P^,0);
    ReadLn;
    CloseGraph
END.
```

Пояснения: Чтобы скопировать область экрана, Паскаль должен сначала ее запомнить в оперативной памияти. Выберем мысленно прямоугольник, охватывающий объект, предназначенный для копирования. В нашем случае подойдет прямоугольник между точками (10,60) и (120,140). Чтобы знать, сколько памяти отвести под запоминание области, компьютер должен знать размер изображения в байтах. Этот размер сообщает функция ImageSize. Поскольку размер этот может оказаться большим, то запоминать изображение лучше всего в куче. Отводит место в куче процедура *GetMem*. Вот ее вызов - *GetMem(P, Size)*. *P* – так называемый **указатель** на место в памяти, предназначенное для запоминания. Указатели – это новый для нас тип данных. Останавливаться на них я не буду, скажу только, что они очень похожи на ссылки. Запоминает область процедура *GetImage*, параметр которой *P^* имеет значением изображение этой области. Процедура *PutImage* помещает это изображение в точно такой же прямоугольник экрана с верхним левым углом в точке (500,400).

Если вы уже запустили эту программу, то могли видеть, что *GetImage* прихватил в выделенном прямоугольнике и кусок диагонали, а *PutImage* добросовестно поместил на экран все, что прихватил *GetImage*, начисто стерев все, что там было раньше. Ответственность за это несет последний параметр *PutImage*, равный у нас нулю. Для того, чтобы новое изображение не затирало старое, нужно использовать вместо нуля двойку.

Задание 130: Нарисуйте шеренгу из десятка снеговиков.

Пример 2. Попробуем двигать наши очки слева направо.

```
USES      Graph,CRT;
VAR       x,Gd, Gm    : Integer;
          P           : pointer;
          Size        : Word;
BEGIN
    Gd := 0; InitGraph(Gd, Gm, 'c:\tp\bgi');
    SetLineStyle(0,0,Thickwidth);
        {Рисуем очки:}
    Circle(50,100,20);    Circle(50,100,15);
    Circle(90,100,20);    Circle(90,100,15);
        {Рисуем диагональ :}
    Line(0,0,640,480);
    Size := ImageSize(10,60,120,140);
    GetMem(P, Size);
    GetImage(10,60,120,140,P^);
        {Начинаем движение:}
    x:=20;
    while x<300 do begin
        PutImage(x, 150, P^,1);
        Delay(40);
        PutImage(x, 150, P^,1);
        x:=x+1;
    end{while};
    CloseGraph
END.
```

Пояснение: Чтобы нарисовать очки в каком-то месте, а потом их стереть, достаточно два раза подряд употребить оператор *PutImage(x, 150, P^,1)*. Обратите внимание, что изображение прямой, по которому прошлись очки, не затерлось. Все это - результат удивительного действия константы *1*. Но чтобы понять механизм ее действия, нужно знать азы алгебры логики, которые, к сожалению, у меня нет времени излагать.

Задание 131: Пусть два снеговика идут друг другу навстречу.

15.6. Вывод текста в графическом режиме

Процедура WriteLn печатает маленькие буквы скучного начертания. В модуле Graph имеются процедуры *SetTextStyle* и *OutTextXY*, которые общими усилиями печатают буквы разного размера (в том числе и очень крупного) нескольких изящных очертаний, причем процедура *SetTextStyle* задает стиль и размер букв, а процедура *OutTextXY* печатает строку букв заданного стиля и размера в заданном месте экрана. Например, в результате выполнения фрагмента

```
SetTextStyle (4,0,8);
OutTextXY(200,300,'Hello!')
```

на экране появится строка *Hello!*, выполненная шрифтом 4 (готический) размера 8. Левый верхний угол строки будет находиться в точке (200,300). Число 0 означает обычное горизонтальное направление текста, 1 – вертикальное, 2 - горизонтальное с лежащими буквами. Цвет текста, как и цвет фигур, определяется процедурой SetColor.

Всего в Паскале есть 5 стандартных шрифтов:

0 - обычный растровый (остальные - векторные)
1 - полужирный
2 - тонкий
3 - газетный
4 – готический

Размер букв зависит от шрифта. Можно независимо менять высоту и ширину букв. Для этого существует процедура *SetUserCharSize*. Например, *SetUserCharSize(7,3, 9,4)* устанавливает ширину букв в 7/3 раза больше нормальной, а высоту - в 9/4.

К сожалению, стандартный набор векторных шрифтов Паскаля не поддерживает русские буквы. Если вы хотите красиво писать по-русски, вам придется где-нибудь найти подходящие файлы русских шрифтов.

Недостаток *OutTextXY* - она выводит только строковые выражения. Покажу, как обмануть ее и напечатать значение числового выражения. Пусть переменная *a* описана, как Integer, и равна 937. Оператор *OutTextXY(200,200, a)* не захочет ее печатать. Тогда придумаем переменную *as* и опишем ее, как String. Следующий фрагмент делает дело:

```
a:=937;
Str(a,as);
OutTextXY(200,200, as)
```

Процедура *Str* преобразует число a=937 в строку из трех символов as='937'.

Для симметрии покажу и обратную процедуру *Val*:

```
VAR  a,err   :Integer;
     as      :String;
BEGIN
   as:='937';
   Val(as,a,err);
   WriteLn (a+1);
   ReadLn
END.
```

Пояснения: Процедура *Val* преобразует строку из трех символов 937 в число 937. В результате оператор *WriteLn (a+1)* печатает число 938. На смысле переменной *err* останавливаться я не буду.

Задание 132: У вас имеется текстовый файл из произвольного числа строк. Организуйте вывод этого файла на экран векторным шрифтом. Если у вас есть время, организуйте управление с клавиатуры: пролистывание текста, выбор шрифта, выбор цвета шрифта и фона.

15.7. Управление цветом в текстовом режиме (модуль CRT)

Если вам не нужно рисовать на экране картинки, то вам не нужен и графический режим. Тем более, что вывод текста в текстовом режиме можно сделать достаточно красиво и удобно. Делается это с помощью модуля *CRT*. Вот какие процедуры он предлагает для этого:

TextColor (Yellow)	Выбор цвета текста
TextBackground (Blue)	Выбор цвета фона под текстом. Я имею в виду цвет не всего экрана, а той узенькой полоски, на которой появляется текущий текст
ClrScr	Очистка экрана. Если в программе предварительно была выполнен оператор *TextBackground (Blue)*, то после выполнения *ClrScr* весь экран станет синим
GotoXY(61,14)	Поместить курсор в 61 столбец 14 строки

До сих пор вы не умели управлять положением текстового курсора на экране. При помощи процедуры **GotoXY** вы сможете помещать курсор в произвольную точку экрана, а значит и текст печатать в произвольном месте экрана. Вспомним, что в текстовом режиме экран обычно разделен на 25 строк по 80 столбцов. Строки пронумерованы сверху вниз, столбцы - слева направо. Чтобы поместить курсор в 61 столбец 14 строки, достаточно записать *GotoXY(61,14)*.

Задача: Закрасить экран красным цветом и посредине экрана желтыми буквами на синем фоне написать “Вход”.

Программа:

```
USES CRT;
BEGIN
   TextBackground (Red);
   ClrScr;
   TextColor (Yellow);
   TextBackground (Blue);
   GotoXY(38,13);
   WriteLn('Вход')
END.
```

15.8. Работа с датами и временем (модуль DOS)

Каждый компьютер имеет часы и календарь. Каждый пользователь может спросить у компьютера, сколько времени и какое сегодня число, а при желании и подправить то и другое. Для работы с временем и датой на Паскале необходим новый для вас **модуль DOS**. Мы рассмотрим четыре процедуры этого модуля, которые работают с временем и датой:

GetTime	Узнать время
SetTime	Установить время
GetDate	Узнать дату
SetDate	Установить дату

Чтобы воспользоваться этими процедурами, вы должны придумать имена переменным, обозначающим дату и время:

God	год (с 1980 по 2099)
Mes	месяц (1-12)
Den	день месяца
Den_Ned	номер дня недели (от 0 (воскресенье) до 6 (суббота))
Chas	час
Min	минута
Sec	секунда
Sotki	сотые доли секунды

Все эти переменные должны быть целочисленными, однако не типа Integer, к которому вы привыкли, а типа **Word**. Таково требование упомянутых процедур. Переменная типа Word должна быть целым числом из диапазона 0 - 65535.

Вот программа, которая узнает у компьютера, какое сегодня число и сколько времени:

```
USES DOS;
VAR  God, Mes, Den, Den_Ned,    Chas, Min, Sec, Sotki     : Word;
BEGIN
   GetDate(God, Mes, Den, Den_Ned);
   WriteLn('Сегодня       ' , Den,  ' . '   ,Mes,   '   '  ,God,   '  года');
   GetTime(Chas,Min,Sec,Sotki);
   WriteLn('Сейчас        ' , Chas,' час.   '    ,Min, ' мин.  и   ' ,Sec,  '  сек.')
```

```
END.
```

Задание 133 «Быстрота реакции»: Определите быстроту своей реакции: На экране через случайный промежуток времени (секунды через 2 - 4) возникает квадрат. Как только он возник, быстрее нажимайте на какую-нибудь клавишу. Пусть компьютер вычислит, сколько сотых долей секунды прошло между этими двумя событиями.

Возможно, вас не удовлетворит работа *GetTime* в этой программе. Попробуйте сделать определитель быстроты реакции, не связываясь с процедурами модуля DOS. Например, засеките по секундомеру, сколько времени ваш компьютер выполняет пустой цикл *for i:=1 to 10000000 do;* и создайте на этой основе свою единицу времени.

А вот как установить дату и время:

SetDate(1997, 5, 22) -установить дату 22 мая 1997 года

SetTime(23, 58, 32, 93) -установить время 23 часа 58 мин 32 сек и 93 сотых

Имейте в виду, что эти процедуры устанавливают дату и время не только для Паскаля, а для всего компьютера. Будьте осторожны, а не то файлы, сохраненные вами, будут иметь неверный атрибут времени.

Задание 134 «Определитель дня недели»: Вы вводите любую дату. Компьютер должен определить день недели и напечатать его в виде "*понедельник*", "*вторник*" и т.д. При этом вы не должны испортить календарь, то есть после того, как ваша программа завершит работу, календарь компьютера должен показывать правильную дату, а не ту, что вы ввели.

15.9. Нерассмотренные возможности Паскаля

Вы узнали о Паскале самое основное и распространенное. Однако это составляет, дай бог, одну третью часть всех богатств Паскаля. Оставшиеся 2/3 не изложены мной, так как они или менее часто употребляются, или слишком сложны для вводного курса. Еще одна причина - многие из них имеют дело с низкоуровневыми и чувствительными ресурсами компьютера, а современные компьютеры и операционные системы имеют тенденцию запрещать программисту прямое обращение к этим ресурсам. Поэтому я ограничусь кратким обзором этих возможностей. Интересующихся же я отсылаю к книге Полякова, указанной в списке литературы.

Модуль Graph. Подобно курсору в текстовом режиме, в графическом режиме есть свой **графический курсор** (невидимый). Существует несколько процедур, удобно рисующих с помощью этого курсора отрезки и ломаные. Имеется возможность использовать больше, чем 16 цветов. Можно использовать для заливки ваши собственные узоры. Можно использовать так называемые **видеостраницы**, которые ускоряют вывод на экран графической информации. Можно организовать на экране несколько **графических окон**, в каждом из которых в независимой системе координат отображать свою графическую информацию. Можно более тонко управлять выводом на экран векторных шрифтов, устанавливать собственные шрифты. Можно устанавливать разные видеорежимы и использовать свои видеодрайверы.

Модуль CRT. Подобно модулю Graph, организующему графические окна, модуль CRT может организовывать **текстовые окна**, может менять количество букв в строке с 80 на 40, раздвигать и сдвигать текстовые строки, менять яркость текста или делать его мигающим.

Модуль Overlay. Если в результате компиляции вашей длинной паскалевской программы программа на машинном языке получается очень большая, она может и не уместиться в оперативной памяти компьютера, а значит и не сможет выполниться. Модуль Overlay позволяет откомпилировать вашу программу по частям, каждую часть записав в отдельный исполнимый файл. Теперь при выполнении откомпилированной программы, если память забита, в нее не загружаются те части программы, которые в данный момент не нужны, а когда в них возникает необходимость, из памяти выгружаются, чтобы освободить им место, отработавшие части.

Файлы. Кроме текстовых файлов Паскаль различает еще два типа файлов: типизированные и бестиповые. Типизированный файл Паскаль рассматривает, как цепочку данных определенного типа (например, чисел типа Integer или записей заданной структуры). Бестиповой файл Паскаль рассматривает, как длинную цепочку битов, предназначенную для ввода в определенное место оперативной памяти, или наоборот – полученную в результате записи определенной области оперативной памяти на диск.

При работе с файлами и каталогами Паскаль позволяет:

- создавать, переименовывать и уничтожать файлы
- создавать и уничтожать каталоги
- определять текущий каталог

- устанавливать текущий каталог

Модуль DOS согласно своему названию позволяет программисту использовать в паскалевской программе многие возможности и команды операционной системы MS-DOS, а именно:

- определять размер дисков и свободного места на диске
- искать файлы
- определять их атрибуты
- анализировать полное имя файла (дорожку)
- работать с прерываниями DOS
- организовывать субпроцессы и резидентные программы

Ключи компиляции. В 15.2 мы уже рассматривали ключ компиляции {$I…….}, который позволял включать в программу в качестве фрагмента программный текст из другого файла. Для других целей существуют другие ключи компиляции, которые называются так потому, что используются Паскалем во время компиляции для настройки работы компьютера с программой и для настройки режима компиляции. Эти ключи можно узнать по конструкции {$λ…….}, где вместо λ стоит латинская буква.

Процедуры и функции. Паскаль позволяет использовать процедуры и функции, написанные на языке Ассемблера и предварительно откомпилированные.

Паскаль позволяет хранить процедуры и функции в памяти не только, как составные части программы, но и как данные. Для этого используется специальный тип данных – процедурный. Имена процедур и функций могут служить параметрами в списках параметров других процедур и функций. Это позволяет организовать удобную математическую обработку данных.

Математические возможности Паскаля включают действия над двоичным представлением данных.

Объекты. Введение процедур в программирование резко повысило надежность создаваемых больших программ и их обозримость, сделало сам процесс программирования более удобным. Следующим шагом на пути совершенствования программирования стало введение объектов. **Объект** – это синтез данных и процедур, которые эти данные обрабатывают. Структура объекта такая же, как и у записи (record), только вдобавок к полям для данных имеются поля для процедур и функций. Вот пример записи типа объекта:

```
TYPE X1 =     OBJECT
                    A       :Integer;
                    B       :String;
                    Procedure C(f:Char);
                    Function D:Word
              END
```

Объекты в программировании напоминают объекты реального мира. Например, чтобы описать стенные часы, мы должны описать совокупность их составных частей (шестеренки, маятник и прочее – в общем, «данные») плюс совокупность процессов взаимодействия этих частей (как качается маятник, как шестеренка цепляет шестеренку и так далее – в общем «процедуры и функции»).

Типичный пример объекта в программировании – окно в программе Windows. Чтобы заставить окно на экране функционировать, как надо, программисту пришлось описать его размер, цвет, толщину рамки и прочее (данные) плюс процессы перетаскивания его по экрану, изменения размера и прочее (процедуры и функции).

Низкоуровневое программирование. Наиболее эффективные программы пишутся на языке низкого уровня Assembler, но на этом языке неудобно создавать большие программы. Профессиональный программист знает, какой участок его паскалевской программы наименее эффективен (например, выполняется медленно). Он может записать этот участок на Ассемблере и вставить в паскалевскую программу. Более того, он может вставить в паскалевскую программу участок на машинном языке.

Вы можете работать не с переменными величинами, а непосредственно с участками оперативной памяти и видеопамяти компьютера, задавая их адреса в шестнадцатеричной системе счисления. Все это увеличивает эффективность работы программы, но лишает вас защитных механизмов Паскаля – в случае малейшей ошибки ваша программа зависнет, и никто не скажет вам почему.

15.10. Миг между прошлым и будущим

Вот и все. На этом изложение программирования на Паскале я заканчиваю. Того, что вы знаете, вполне достаточно для программирования любых задач из любой сферы человеческой деятельности. Дальнейшее изучение Паскаля позволит вам составлять более эффективные программы, имеющие новые возможности в смысле использования объектов, богатства цветов, быстрой работы с графикой и т.п.

Если вы решите стать профессиональным программистом, то будете работать с современными версиями языков программирования, такими как Delphi, и узнаете, что они полностью объектно-ориентированы и имеют бо-

гатый набор «модулей», предназначенных для удобного решения часто встречающихся типов задач, таких, например, как управление базами данных, создание окон и кнопок на экране и т.п. По сути, настолько удобного решения, что программу для этих задач компьютер пишет сам, а программист только задает исходные данные. Однако, писать программный текст вам придется все равно.

А теперь я хочу предложить вам ***задание 135 и последнее*** на звание *«Программист-любитель II ранга»*. На выбор – одно из трех:

- Игра в морской бой
- Игра в крестики-нолики на бесконечном поле
- Игра в шашки

Во всех трех программах игра должна вестись между человеком и компьютером.

Правила морского боя и шашек общеизвестны. Правила крестиков-ноликов на бесконечном поле такие же, как и у крестиков-ноликов на поле 3 на 3, с тем отличием, что в линию нужно выстроить не 3, а 5 ноликов или крестиков. Конечно, запрограммировать игру на бесконечном поле довольно трудно, поэтому рекомендую ограничиться полем 20 на 20 или 10 на 10.

Требования и рекомендации к программам:

- Компьютер должен обнаруживать незаконное расположение кораблей, незаконные ходы в крестики-нолики и в шашки.
- Компьютер должен вести счет партий и отображать его на экране
- Компьютер должен обеспечить возможность сохранения игры и загрузки сохраненной игры
- Удобный интерфейс. В частности, человек должен иметь возможность легко делать ходы шашками, расставлять корабли, ставить нолики или крестики (например, при помощи клавиш передвижения курсора и клавиши пробела)
- При нажатии на клавишу F1 любая порядочная программа предлагает помощь. В вашем случае достаточно показать правила игры
- Неплохо сделать в углу экрана управляемое с клавиатуры меню с такими, примерно, пунктами: *сохранить игру, загрузить игру, выход из игры, помощь*.
- Для того, чтобы не было игр-близнецов, ходы компьютера не должны быть железно заданы. Например, свои корабли компьютер должен располагать от игры к игре с разумной долей случайности, чтобы человек не мог легко догадаться, где будут стоять корабли в следующей игре. То же относится к выстрелам, ходам в крестики-нолики и в шашки.
- Рекомендации по выбору уровня сложности стратегии: В *морском бое* стратегия компьютера должна быть очень сильной, чтобы человеку было трудно у него выиграть. В *крестиках-ноликах* стратегию компьютера затруднительно сделать очень сильной. Достаточно, если компьютер будет обнаруживать простейшие угрозы человека: четверки с одним свободным концом и тройки с двумя свободными концами - и сам стремиться к их созданию. Что касается *шашек*, то шашечного чемпиона сделать очень трудно. Достаточно, если компьютер не будет бестолку подставлять свои шашки под бой. Правила шашек упростите – например, откажитесь от дамок.

Любая из этих задач достаточно сложна и потребует многих дней напряженной работы. Если вам кажется, что вы не сможете удовлетворить всем перечисленным в задании требованиям, потому что «мы этого не проходили», то я вам заявляю – все, что нужно, мы проходили! Нужно только немножко подумать.

У вас есть все шансы сделать так, что программа в морской бой будет играть очень сильно, а значит, может приобрести популярность во всем мире, так как при выполнении всех требований задания у нее будет достаточно «товарный» вид. В принципе, если вы очень постараетесь со стратегией, то сможете достигнуть того же и в крестики-нолики.

Желаю успеха!

Часть IV. Работа в Паскале на компьютере

В этой части мы не будем изучать программирование. Мы будем учиться, как написанные заранее программы вводить в компьютер, отлаживать и выполнять.

Что нужно знать и уметь к моменту выполнения первой программы?

- Вы должны иметь работающий компьютер с установленным на жестком диске Паскалем. Вы должны знать, где у компьютера кнопка включения. Очень желательно, чтобы кто-то опытный сказал вам, что Паскаль работает нормально, и показал, как его запускать. Если опытных рядом нет, я с грехом пополам попробую рассказать вам, как запустить Паскаль («с грехом пополам» потому, что для запуска Паскаля нужно иметь минимальные навыки работы или в Windows или в Norton или в чем-нибудь подобном, а это тема отдельной книги).
- Вы должны уметь работать с текстом в простейшем текстовом редакторе. Если не умеете, то вам вполне достаточно изучить приложение П1, где вам будут даны все необходимые навыки.
- Считаю совершенно необходимым, чтобы вы сохраняли все ваши программы, даже самые маленькие, на диске. Поэтому вы должны заранее знать, что такое каталоги (папки) и файлы. Если не знаете, то вам вполне достаточно прочитать приложение П2, где вы узнаете все необходимое.
- Вы должны знать имя каталога Паскаля и дорожку к нему (что это такое, читайте там же – П2).

Порядок работы в Паскале

(1) Запустите Паскаль
(2) Введите программу
(3) Сохраните программу на жестком диске
(4) Выполните программу
Если результаты вас удовлетворяют, перейдите к пункту 6
(5) Исправьте ошибки в программе
Вернитесь к пункту 3
(6) Отдыхайте

Рассмотрим подробно и по порядку пункты этого алгоритма.

(1) Запуск Паскаля

Итак, вы в отчаянном положении. Вы хотите запустить Паскаль, но не знаете, как это сделать, и некому вам объяснить. Я сейчас попробую вам помочь, но шансы на успех – 50 на 50.

Прежде всего прочитайте приложение П2 и параграфы (2) и (3) из этой главы, так как они помогут вам путешествовать по каталогам (обратите, кстати, внимание на двойную точку (..)). Напомню, что вы должны знать

имя каталога Паскаля на вашем компьютере и дорожку к нему. Если не знаете, то могу вам подсказать, что скорее всего имя каталога – *TP* или *TP70* или *PASCAL* или что-то очень близкое (если у вас Borland Pascal, то, возможно, *BP*). А находится этот каталог скорее всего в корне одного из логических дисков. Если его нет в корне, то он – внутри какого-нибудь каталога, скажем, *PROGRAMS*. Только не ищите его в каталоге *WINDOWS*, это опасно, да и не бывает его там. Вы должны также знать имя запускающего файла в каталоге Паскаля. Это *turbo.exe* (если у вас Borland Pascal, то *bp.exe*). Он может находиться не в самом каталоге Паскаля, а в его подкаталоге *BIN*.

А теперь вперед:

Включите компьютер, если он еще не включен. Некоторое время по черному экрану бегут белые слова – это компьютер рапортует вам о том, как он сам себя проверяет, загружает операционную систему (ОС) и, возможно, другие программы. Затем на экране устанавливается и замирает картинка. Ее вид зависит от ОС и других загруженных программ. В подавляющем большинстве случаев возможны три варианта:

А) Независимо от ОС, у вас автоматически загружается операционная оболочка Norton Commander. Это было бы легче всего.

Б) Ваша ОС – Windows 95.

В) Ваша ОС – Windows 3.1 или 3.11.

Рассмотрим все эти три варианта.

А) Загружен Norton Commander. Его вы узнаете по синему экрану, разделенному по вертикали пополам на две “панели”. На каждой панели – список файлов и подкаталогов какого-нибудь каталога какого-нибудь логического диска вашего компьютера. Имена каталогов приведены заглавными буквами, а имена файлов с расширениями – строчными. Обе панели совершенно равноправны. Это просто два независимых “окна”, глядящие внутрь ваших дисков. Наверху каждой панели написана дорожка к каталогу, внутренность которого вы лицезреете на панели.

Для начала щелкните по клавише *F2*. Возможно, в открывшемся меню вы сразу найдете Паскаль. Если нет, то при помощи клавиши *Esc* уберите меню с экрана.

При помощи клавиш ← , → , ↑ , ↓ , *Tab, Enter* попробуйте добраться до каталога Паскаля на манер того, как я вам советую в параграфе (3) добраться до вашего каталога. Только вместо полос прокрутки пользуйтесь мышкой или клавишами ↑ , ↓, ← , → , а вместо двойного щелчка мышкой можете пользоваться клавишей *Enter*. Если вы хотите посмотреть содержимое другого логического диска, то, удерживая нажатой клавишу *Alt*, щелкните по клавише *F1* или *F2* и из появившегося меню выберите нужный диск.

Вот вы нашли Паскаль. Зайдите внутрь каталога Паскаля. Здесь вам нужно найти запускающий файл. Запустите Паскаль, щелкнув по этому файлу клавишей *Enter*. Дело сделано.

Б) Загружена ОС Windows 95. Попробуйте найти на экране значок с надписью *Turbo Pascal* или близкой, и если нашли, щелкните по ней мышкой дважды. Если двойной щелчок не получается, щелкните один раз, а затем нажмите клавишу *Enter*. Дело сделано.

Если значок не нашли, найдите в левом нижнем углу экрана «кнопку» *Пуск* или «подковырните» ее оттуда мышкой. Щелкните по ней мышкой. Щелкните в выпавшем меню слово *Программы*, а там – *Проводник*. Откроется окно, в котором вы попробуйте добраться до каталога Паскаля на манер того, как я вам советую в параграфе (3) добраться до вашего каталога. Но рычаги управления здесь другие. В левой половине окна Проводника вы увидите дерево, похожее на то, что я описывал в П2. Пользуясь полосами прокрутки, найдите на нем значок нужного вам логического диска и если слева от него стоит значок “+”, щелкните по плюсу. Из логического диска “выскочат” ветки-каталоги. Щелкните по значку нужного каталога. В правой половине окна Проводника вы увидите список файлов и подкаталогов этого каталога. Каталоги изображены значками в виде желтых папочек, файлы – любыми другими. Найдите запускающий файл в виде белого прямоугольника с нужным именем и щелкните его дважды.

Может так случиться, что Паскаль заартачится, не захочет запускаться, начнет предупреждать и жаловаться на что-то. Мой совет – попробуйте запустить Norton Commander. Для этого повторите все описанные действия, но не для запуска Паскаля, а для запуска Нортона. Его запускающий файл – *nc.exe*, находится он в каталоге, имя которому, скорее всего, *NC* или *NC50* или *Norton* или близкий. Затем перейдите к пункту А).

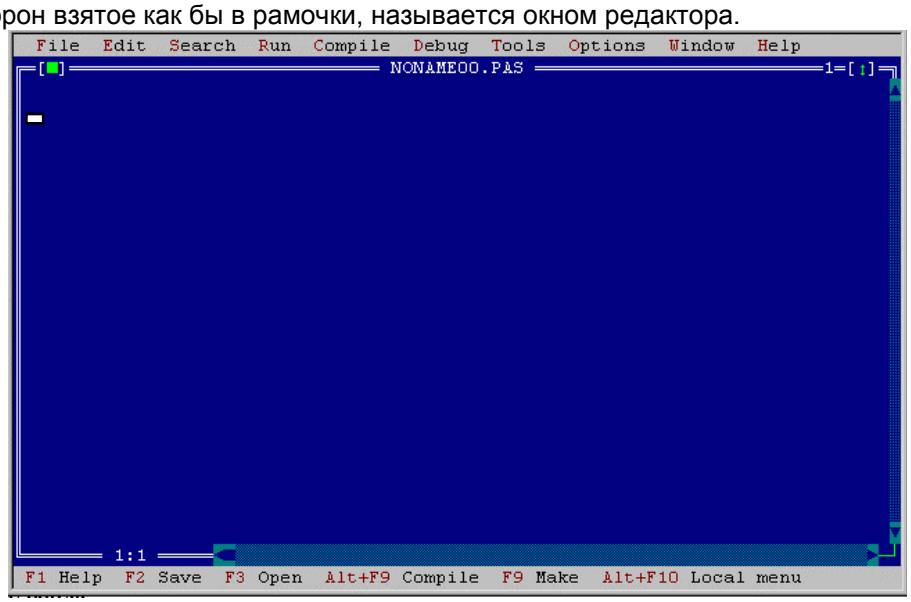
В) Загружена ОС Windows 3.1 или 3.11. Перед вами – окно с надписью “Диспетчер программ”. Если окна нет, значит – значок с такой надписью. Щелкните по нему дважды, чтобы он превратился в окно. Если двойной щелчок не получается, щелкните один раз, а затем щелчок по слову «Развернуть». Окно раскрылось.

Щелкните по слову “Файл”. В выскочившем из него меню – по слову “Выполнить”. В узеньком окошке с названием “Командная строка” мигает курсор. Наберите на клавиатуре дорожку до Паскаля (например, c:\tp\turbo.exe) и щелкните по клавише ввода. Если дорожка верна, то дело сделано. Если нет, то Windows сообщит вам, что файл она не нашла. Щелкните по сообщению “ОК” на экране и попробуйте найти Паскаль, щелкнув на экране по кнопке “Пролистать”. Выберите мышкой “устройство” (нужный диск). Затем попробуйте найти “каталог”, щелкая дважды по значкам кажущихся вам подходящими каталогов. Начните с самого верхнего значка. Если двойной щелчок не получается, щелкайте один раз, а затем щелчок по слову «ОК». При этом в левом окне вни-

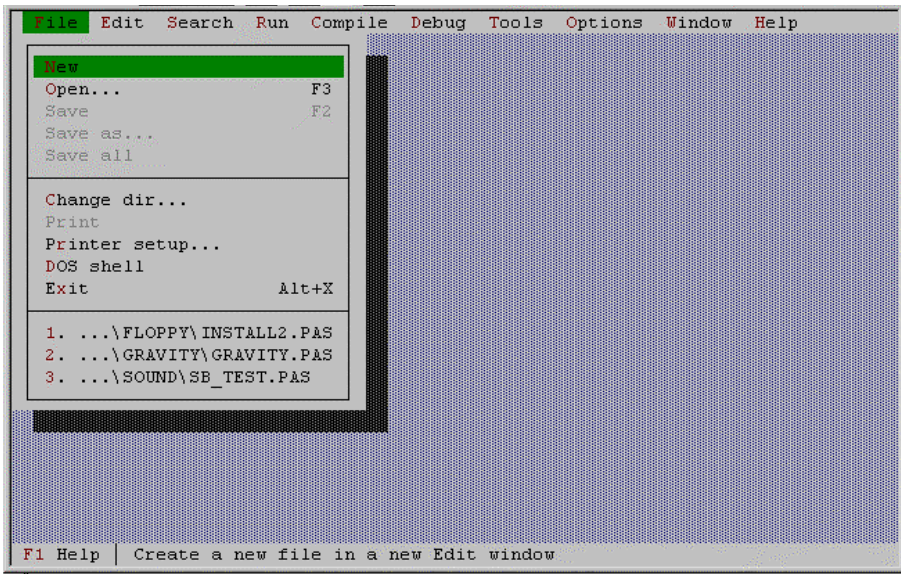
мательно просматривайте имена файлов, входящих в выбранный вами каталог. В окне “Каталоги” значки каталогов-родителей расположены чуть левее, а каталогов-“детей и внуков” – правее. Пользуйтесь прокруткой. Заметив в левом окне запускающий файл Паскаля, щелкайте по нему – и ОК. И еще раз ОК. Дело сделано.

(2) Начало работы. Ввод программы. Выход из Паскаля

Итак, вы запустили Паскаль и первое, что вы увидели, это пустой синий экран. В верхнем левом углу мигает черточка – курсор. Значит, Паскаль приглашает вас вводить текст программы. Это пустое синее пространство, со всех сторон взятое как бы в рамочки, называется окном редактора.



Иногда Паскаль настроен так, что при запуске вместо окна редактора показывает бессмысленное серое пространство - рабочий стол. Курсора нигде не видно, программу вводить некуда. Не огорчайтесь - дело легко поправить. Поправить или мышкой или с клавиатуры.



Мышкой быстрее - щелкните по слову *File* в главном меню наверху экрана, а затем в выпавшем из него другом меню - по слову *New*.

Если с клавиатуры, то нажмите клавишу *F10*, затем клавишами ← или → подсветите слово *File* и нажмите клавишу ввода. Затем клавишами ↑ или ↓ подсветите слово *New* и снова нажмите клавишу ввода.

Можно вводить программу. Напоминаю, что вводится она, как обычный текст в простейшем текстовом редакторе. Если вы этого не умеете, то вам вполне достаточно изучить приложение П1, где вы получите все необходимые навыки.

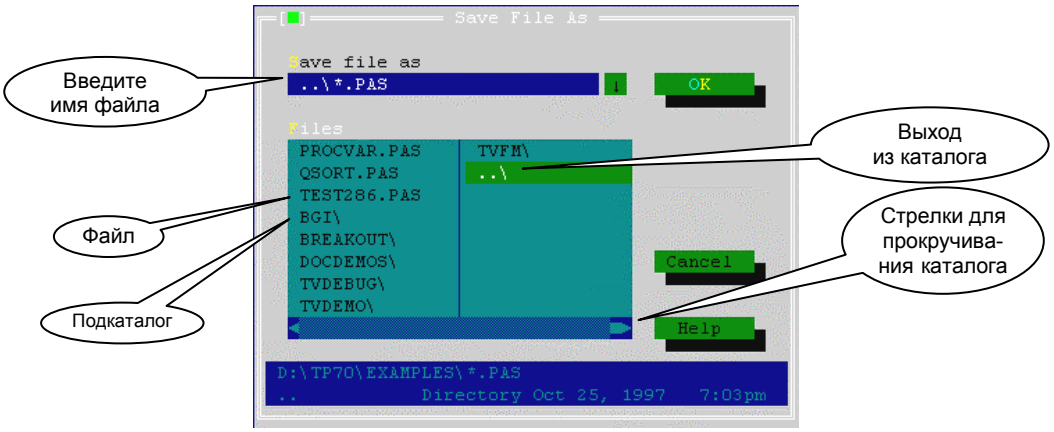
(В дальнейшем для краткости я не буду писать, как выбирать пункты меню. Я просто напишу **File** → **New**).

Выход из Паскаля - **File** → **Exit.**

(3) Сохранение программы на диске. Загрузка программы с диска

Итак, у вас на экране программа, готовая к выполнению. Во время выполнения может произойти неприятная вещь – компьютер может зависнуть. Это означает, что изображение на экране замрет, и никакими клавишами или мышкой вы не сможете вывести компьютер из ступора. Придется компьютер выключать и вновь включать. Поскольку ваша программа находится пока только на экране и в оперативной памяти, то при выключении компьютера она пропадет и вам придется вводить ее снова. Чтобы избежать лишней работы, вы должны перед выполнением программы записать (сохранить) ее на диск.

Для этого в меню *File* есть опция *Save*. Как только вы ее выберете, перед вами появится диалоговое окно – Паскаль предлагает вам выбрать каталог, в который вы хотите записать программу, и имя, которое вы хотите дать файлу вашей программы (Если вы не очень хорошо знаете, что такое файл и каталог, то сейчас самое время прочесть приложение).



Если вас не устраивает каталог, который предлагает Паскаль, то пролистайте его, щелкнув по стрелкам полосы прокрутки, чтобы найти две точки (..), которые служат выходной дверью из каталога. Щелкните по ним дважды – и вы окажетесь в родительском каталоге. Интерфейс Паскаля менее удобен, чем интерфейс Windows, но вы сможете все-таки отличать имена подкаталогов от имен файлов по косой черте (BackSlash) в конце имени подкаталога.

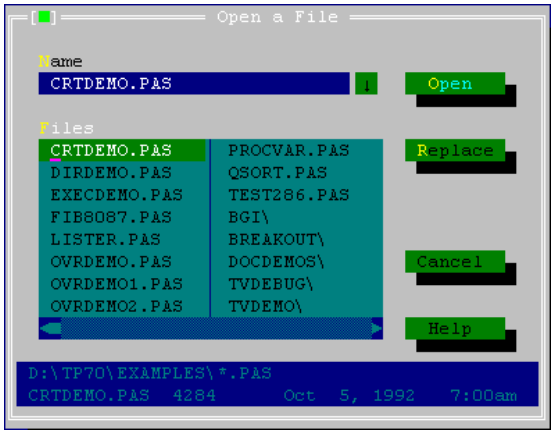
Путешествуя подобным образом по каталогам, вы в конце концов дойдете до нужного. Дважды щелкните по нему и вы окажетесь внутри него. После этого щелкните внутри поля для имени файла. Там замигает курсор. Сотрите все, что там написано, и введите туда придуманное вами имя файла. После этого нажмите клавишу ввода или щелкните на экране по кнопке *OK*. Ваша программа записана на диск. В качестве подтверждения заголовок вашего листа изменился с *NONAME00.PAS* на имя вашего файла с расширением *PAS*.

Теперь программу можно запускать на выполнение.

После выполнения программу обычно исправляют и дополняют и перед следующим выполнением опять сохраняют: **File** → **Save**. Но в этот раз диалоговое окно не появляется. Паскаль, ни о чем вас не спрашивая, стирает с диска всю вашу старую программу и на ее место записывает с тем же именем ту, что вы видите на экране (и то, что вы видите, прокрутив окно, конечно). Так поступают все современные программные продукты.

Если старый вариант программы вам по каким-то соображениям дорог и вы не хотите его стирать, вам нужно выполнить **File** → **Save as…** На экране возникает то же самое диалоговое окно, предлагающее вам то же самое имя файла в том же самом каталоге. Измените имя или каталог и щелкните по кнопке *OK*. Ваша программа с экрана будет записана в другое место диска, а старый файл будет нетронут.

Загрузка программы с диска. Предположим, вы вчера сохранили свою программу, а сегодня хотите продолжить с ней работу. Вы включаете компьютер, запускаете Паскаль, перед вами пустое окно редактора. Ваши действия: **File** → **Open**. На экране появляется диалоговое окно, очень похожее на то, что появляется при сохранении программы:



И действия ваши очень похожи: вы должны отыскать в окне файл вашей сохраненной вчера программы, для чего вам придется попутешествовать по каталогам. Найдя файл, щелкните по нему дважды или один раз по нему, один раз по *Open*. Ваша программа появляется на экране.

(4) Выполнение программы

Перед тем, как выполнить программу, убедитесь, что курсор мигает внутри окна этой программы или что это окно обведено двойной, а не одинарной рамочкой. Я об этом упоминаю потому, что если вы работаете с несколькими окнами (о чем речь ниже), то курсор может мигать в любом из них.

Выберите пункт главного меню *Run*. В выскочившем подменю щелкните по опции *Run*. Все остальное делает компьютер. Начинает он с того, что за долю секунды компилирует вашу программу. После этого, не прерываясь ни на мгновение, он начинает выполнять откомпилированную программу. При этом окно редактора уходит с экрана и на нем возникает черное окно пользователя, на котором Паскаль и будет печатать и рисовать все, что прикажет программа. Если программа короткая и не содержит остановок и пауз, то Паскаль мгновенно печатает и рисует все, что приказано, и возвращает на экран окно редактора. Поскольку все произошло мгновенно, то создается впечатление, что ничего и не произошло – на экране было окно редактора, потом что-то мелькнуло, - и снова на экране окно редактора, а результатов никаких и не видно.

Чтобы увидеть результаты, можно пойти двумя путями:

- После того, как программа уже выполнилась, выбрать опцию **User Screen**. В Турбо-Паскале 7.0 она находится в пункте меню *Debug*. Если вам не хочется искать ее в меню, нажмите *Alt-F5*. На экране возникнет окно пользователя с результатами. Этот способ плох при работе в графическом режиме: изображение или искажено, или вообще не просматривается.
- Гораздо удобнее использовать в программе оператор *ReadLn*. В этом случае, наткнувшись во время выполнения программы на *ReadLn*, Паскаль останавливает программу, окно пользователя с экрана не пропадает, и вы можете смотреть на результаты, сколько хотите. Программа продолжит работу с места остановки, когда вы нажмете на клавишу ввода.

Прокрутка окна пользователя. Если ваша программа печатает много результатов, то все они могут и не поместиться на экране. В этом случае Паскаль начинает прокручивать окно пользователя, и самые верхние результаты исчезают из окна (их уж не вернуть!). Поскольку прокрутка обычно идет быстро, то вы просто не успеваете рассмотреть бегущие по экрану и убегающие наверх результаты. Спокойно посмотреть на них вы сможете только при остановке программы (по *ReadLn* или по окончанию работы). Чтобы результаты не успели убежать нерассмотренными наверх, используйте не один, а несколько *ReadLn*, расставив их в нужных местах программы.

Зацикливание. Нормальная программа, выполнив все, что нужно, заканчивает работу. Однако, если вы допустили ошибку и в программе выполняется бесконечный цикл, то программа не завершится никогда. Вы вечно будете смотреть на экран, по которому бесконечно бегут непонятные числа или слова или рисуются графические объекты, а возможно и ничего не происходит, экран пустой – все зависит от характера ошибки.

Для прерывания работы программы (в том числе и зациклившейся) существует комбинация клавиш *Ctrl-Break*. На экран возвращается окно редактора. Строка программы, на которой она была прервана, выделяется полосой белого цвета. Если вы снова запустите программу, она продолжит работу с прерванного места. Чтобы начать сначала, уберите полосу с экрана клавишами *Ctrl-F2*.

(5) Исправление ошибок. Отладка программы.

Сообщения об ошибках. Ошибки в программах делятся на те, которые Паскаль замечает, и на те, что не замечает и в принципе заметить не может. К первым относятся все синтаксические погрешности, например, *BIGIN* вместо *BEGIN*. Их Паскаль замечает еще на стадии компиляции. На стадии выполнения он замечает такие ошибки, как *Sqrt(-25)*, то есть квадратный корень из -25. Но вот, если вы, желая возвести число *a* в куб, вместо *a*a*a* пишете *a*a*, то этого не заметит ни один язык в мире.

Обнаружив грамматическую ошибку, Паскаль выдает золотыми буквами на красном фоне краткое описание ошибки и ставит курсор в то место программы, где, по его мнению, она находится.

Обнаружив ошибку на стадии выполнения, Паскаль выдает белыми буквами на черном фоне окна пользователя сообщение *Runtime error* и иногда золотыми буквами на красном фоне краткое описание ошибки и ставит курсор в то место программы, где, по его мнению, она находится.

Вот наиболее типичные для начинающих сообщения об ошибках того и другого рода:

Сообщение	Перевод	Вероятная причина ошибки
Unexpected end of file	Неожиданный конец файла	Вы забыли поставить точку после последнего *END*. Или не совпадает количество *begin* и количество *end*
“;” expected	Ждал точку с запятой	Вы забыли поставить точку с запятой после предыдущего оператора
“,” expected	Ждал запятую	Вы указали слишком мало параметров в обращении к подпрограмме
“)” expected	Ждал скобку “)”	Вы указали слишком много параметров в обращении к подпрограмме
Unknown identifier	Неизвестное имя	Вы забыли описать это имя в разделе описаний Неправильно записали стандартное имя, например, *ReedLn* вместо *ReadLn*
Type mismatch	Несовпадение типов	В вашей программе встречаются примерно такие «сладкие парочки»: *VAR c:String; … c:=1+2* или *VAR h:Integer; … h:=9/7*
Duplicate identifier	Дубль имени	Одно и то же имя описано два раза. Например, *VAR a, c, a :String;*
Syntax error	Синтаксическая ошибка	Паскаль затрудняется назвать причину ошибки. Часто причина в том, что вы забыли взять строковую константу в кавычки
BEGIN expected	Ждал BEGIN	Возможно, не совпадает количество *begin* и количество *end*
END expected	Ждал END	Возможно, не совпадает количество *begin* и количество *end*
String constant exeeds line	Строковая константа превышает допустимую длину	Вы забыли закрыть кавычки в строковой константе
Line too long	Строчка слишком длинна	Слишком длинная строчка в программе (не путать со строковой константой, которую нужно брать в кавычки). Не рекомендую залезать программным текстом за правый край экрана
Disk full	Диск заполнен	На вашем диске не осталось места. Надо что-то стереть
Lower bound greater than upper bound	Нижняя граница диапазона больше верхней	Например, вы вместо *array[2..5]* написали *array[5..2]*.
Invalid floating point operation	Неправильная операция с вещественным результатом	*Sqrt(-25)* или *a/0* или что-нибудь в этом роде
Ordinal expression expected	Ждал выражение порядкового типа	Например, вы вместо *for i:=1 to 8* написали *for i:=1 to 8.5*
Error in expression	Ошибка в выражении	Например, вы вместо *k:=a*8* написали *k:=a**8*
Range check error	Ошибка проверки диапазона	Переменная в процессе выполнения программы вышла за пределы допустимого диапазона, как например, в 12.9
Constant out of range	Константа не в диапазоне	Величина константы в программе превосходит допустимый диапазон
Invalid numeric format	Неправильный числовой формат	Если, например, вы по оператору *ReadLn(k)* в программе *VAR k:Integer; …. ReadLn(k) …* пытаетесь ввести число 25.3

Более подробное описание некоторых ошибок вы найдете в 4.3.

Понять смысл многих других сообщений об ошибках вам поможет перевод некоторых часто встречающихся в сообщениях слов:

expected	*ждал*
identifier	*имя*
invalid	*неправильный*

operation	*операция*
error	*ошибка*
variable	*переменная*

Невозможно создать компилятор, который бы всегда точно находил место и причину ошибки. Поэтому приготовьтесь к тому, что некоторые сообщения вы будете долго разглядывать в недоумении. Считайте их не реальными ошибками, а намеками на реальные ошибки.

Пошаговый режим

Вы запускаете программу, Паскаль на ошибки не жалуется, но результаты вас не удовлетворяют. Предположим, что внимательно просмотрев программу, ошибку вы не обнаружили. Все. Вы в тупике.

На случай возникновения таких ситуаций Паскаль предлагает ряд отладочных средств. Я начну с пошагового режима.

Идея пошагового режима вот в чем. Компьютер слишком быстро выполняет программу и человек не успевает проследить «за ходом его мыслей». Хорошо бы мы сами задавали темп выполнения программы. Тогда ошибку обнаружить было бы значительно легче.

Рассмотрим задачу из 5.1: В компьютер вводятся два произвольных положительных числа - длины сторон двух кубиков. Компьютер должен подсчитать объем одного кубика - большего по размеру . Обозначаем *a1* - сторону одного кубика, *a2* - сторону другого, *V* - объем кубика. Приведем второй вариант программы:

```
VAR a1,a2,V : Real;
BEGIN
  ReadLn (a1,a2);
  if a1>a2
      then V:=a1*a1*a1
      else V:=a2*a2*a2;
  WriteLn (V  : 15:5)
END.
```

Запускайте программу, но не обычным образом, а клавишей **F7**, которая является более быстрым способом вызова меню **Run→Trace into**. (Кстати, обратите внимание, что справа от многих опций меню обозначены «горячие» клавиши, которыми эти опции можно быстро вызывать.) Паскаль откомпилирует вашу программу и сделает паузу перед началом выполнения программы, подсветив горизонтальной полосой строку *BEGIN*.

Еще раз **F7**. Ничего не происходит, только полоса подсветки прыгает на следующую строку. В ней находится первый исполняемый оператор вашей программы – *ReadLn (a1,a2)*.

Итак, правило простое – при нажатии на **F7** Паскаль выполняет одну строку программы и подсвечивает ту строку, которой предстоит быть выполненной.

F7. Паскаль выполняет *ReadLn (a1,a2)*, в результате чего у вас, как и при обычном выполнении программы, во весь экран распахивается черное окно пользователя. Оператор требует ввода двух чисел. Введите, например, 5 и 4, не забыв нажать на клавишу ввода. Окно пользователя пропадет, снова возникнет окно редактора и вы увидите, что серая полоса подсветки перескочила на строку *if a1>a2*.

F7. Паскаль выполняет *if a1>a2*, в результате чего полоса прыгает на *then V:=a1*a1*a1*, так как 5>4.

F7. Полоса перепрыгивает через *else V:=a2*a2*a2* и попадает на *WriteLn (V : 15:5)*.

F7. Паскаль выполняет *WriteLn (V : 15:5)*, в результате чего печатается результат, а полоса перескакивает на строку *END*.

F7. Полоса пропадает. Программа выполнена. Посмотрим на результат при помощи *Alt-F5*.

Теперь еще раз выполните ту же программу в пошаговом режиме. Но когда *ReadLn (a1,a2)* потребует двух чисел, введите 2 и 3. Теперь уже после *if a1>a2* полоса прыгает не на *then V:=a1*a1*a1*, а на *else V:=a2*a2*a2*.

В любой момент пошагового выполнения программы вы можете вместо **F7** выбрать **Run→Run** (клавиши *Ctrl-F9*) и программа продолжит выполняться в обычном режиме. Чтобы прервать пошаговый режим, выберите **Run→Program reset** (клавиши *Ctrl-F2*).

Работа с окнами пользователя и отладчика

Предположим, у вас не получается задание из 6.1 - определить без компьютера, что будет печатать следующая программа:

```
LABEL   m1,met5;
VAR     n,k : Integer;
```

```
BEGIN
        n:=10;
        k:=0;
        WriteLn('Считаем зайцев' );
 met5:Write(n);
        n:=n+k;
        goto m1;
        n:=n+1;
 m1:    Write(' зайцев  ');
        ReadLn;
        k:=k+1;
        goto met5;
        WriteLn('Посчитали зайцев')
END.
```

Ответ: Эта программа должна печатать:

```
Считаем зайцев
10 зайцев
10 зайцев
11 зайцев
13 зайцев
16 зайцев
20 зайцев
……
```

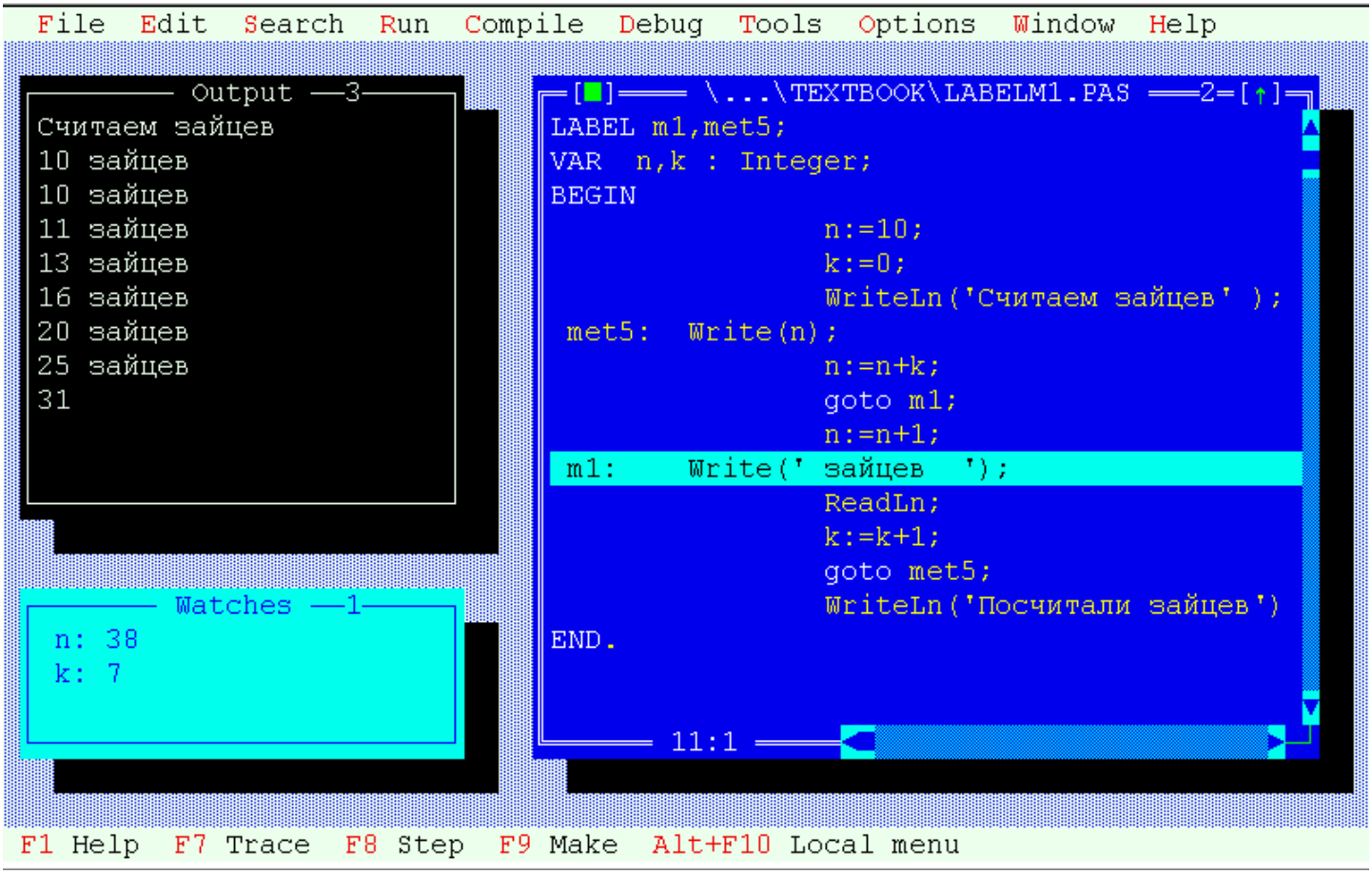
С ответом у вас не сходится. Значит, вы чего-то не понимаете. Чего? Чтобы определить это, вы вводите программу в компьютер. Он печатает все, как в ответе. Почему? Вероятно, вы не понимаете, как изменяются в процессе выполнения программы значения переменных n и k. Хорошо бы можно было подсмотреть за их значениями в оперативной памяти. Хорошо бы также напечатанные результаты не пропадали с экрана, а были бы всегда перед глазами.

Проделайте следующее. (При дальнейшем объяснении я буду предполагать, что вы умеете обращаться с несколькими окнами на экране. Кто не умеет, читайте следующий параграф.)

Сузьте окно вашей программы так, чтобы оно освободило примерно половину ширины экрана. Далее - **Debug→Output**. Перед вами появляется черное окно пользователя, в котором вы привыкли наблюдать результаты, но в уменьшенном виде. Измените его размер и отбуксируйте на свободное место экрана.

Далее - **Debug→Watch**. Перед вами появляется окно отладчика, в котором вы будете наблюдать значения переменных величин в оперативной памяти. Измените его размер и отбуксируйте на оставшуюся часть экрана.

Вот как будет выглядеть ваш экран после всех манипуляций (и уже после нескольких выполнений цикла в пошаговом режиме, о чем дальше):



Вы сами должны сказать Паскалю, за какими переменными хотите следить. Очевидно, в вашем случае это *n* и *k*. Ваши действия: **Debug→Add watch…** ,запишите имя *n* в открывшемся диалоговом окне и – щелчок по кнопке **OK**. В окне отладчика появится сообщение *n:7* или *n:0* или какое-нибудь другое число или сообщение. Не обращайте на это внимания. Повторите всю процедуру с *k*.

Теперь можно запускать программу на выполнение в пошаговом режиме. Для этого нажмите на **F7**. Паскаль откомпилирует вашу программу, возможно, изменит значения в окне отладчика и сделает паузу перед началом выполнения программы, подсветив горизонтальной полосой строку *BEGIN*.

F7. Паскаль, возможно, обнулит значения в окне отладчика. Полоса подсветки прыгает на следующую строку. В ней находится первый исполняемый оператор вашей программы – *n:=10*.

Итак, правило простое – при нажатии на **F7** Паскаль выполняет одну строку программы и подсвечивает ту строку, которой предстоит быть выполненной.

F7. Паскаль выполняет *n:=10*, в результате чего в окне отладчика значение *n* меняется на 10, а серая полоса прыгает на строку *k:=0*.

F7. Паскаль выполняет *k:=0*, в результате чего в окне отладчика значение *k* становится равным 0, а полоса прыгает на строку *WriteLn('Считаем зайцев')*.

F7. Паскаль выполняет *WriteLn('Считаем зайцев')*, в результате чего в окне пользователя появляется напечатанный текст *Считаем зайцев*, а полоса прыгает на строку *met5: Write(n)*.

F7. Паскаль выполняет *Write(n)*. В окне пользователя в новой строке появляется число 10, так как в оперативной памяти *n* равно 10 (об этом нам говорит окно отладчика). а полоса прыгает на строку *n:=n+k*.

F7. Паскаль выполняет *n:=n+k*, в результате чего в окне отладчика значение *n* остается равным 10, так как в оперативной памяти *k* равно 0. Полоса прыгает на строку *goto m1*.

F7. Паскаль выполняет *goto m1*. Полоса перепрыгивает на строку *m1: Write(' зайцев ')*.

F7. Паскаль выполняет *Write(' зайцев ')*, в результате чего в окне пользователя справа от числа 10 появляется текст *зайцев*, а полоса прыгает на строку *ReadLn*.

F7. Паскаль выполняет *ReadLn*, в результате чего окно пользователя распахивается на весь экран и *ReadLn* ждет, когда вы нажмете на клавишу ввода. Вы нажимаете – и полоса прыгает на строку *k:=k+1*.

F7. Паскаль выполняет *k:=k+1*, в результате чего в окне отладчика значение *k* меняется на 11. Полоса прыгает на строку *goto met5*.

F7. Паскаль выполняет *goto met5*. Полоса перепрыгивает вверх, на строку *met5: Write(n)*.

И так далее.

Если ваша программа рисует что-то в графическом режиме, то, к сожалению, при описанном способе работы картинки видны не будут.

Если в вашей программе есть подпрограммы, то серая полоса проникнет и внутрь подпрограмм и будет скакать там, как ни в чем не бывало. Таким образом, вы сможете анализировать и работу подпрограмм. Если же вы не хотите утомлять себя прыжками внутри подпрограмм, нажимайте не *F7*, а *F8*. Тогда Паскаль будет выполнять подпрограмму мгновенно в автоматическом режиме и серая полоса будет путешествовать только внутри основной программы. Вы можете свободно чередовать нажатия *F7* и *F8*. Клавиша *F7* будет направлять вас «вглубь» подпрограмм, а *F8* - «наружу».

Отладка больших программ

Точки прерывания - Breakpoints. Будем считать “большими” программы, превышающие пару размеров экрана. Для них изложенный только что пошаговый режим может быть очень утомительным, так как слишком часто приходится жать **F7**. Вы можете заставить Паскаль при выполнении программы задерживаться не на каждой строке, а только на некоторых - тех, что вам нужно.

Поставьте курсор на одну из этих строк и нажмите *Ctrl-F8*. Строка станет красной. Это знак того, что при выполнении программа на ней остановится. Будем называть эту строку **точкой прерывания** или контрольной точкой.

Поставьте курсор на другую строку и снова нажмите *Ctrl-F8*. Эта строка тоже станет точкой прерывания.

Так проделайте со всеми нужными вам строками. Затем можете обычным образом вывести на экран окна пользователя и отладчика.

А теперь обычным образом (**Run→Run** или *Ctrl-F9*) запускайте программу. Программа будет выполняться до тех пор, пока не наткнется на какую-нибудь точку прерывания. На ней она встанет и продолжит работу с того места, где остановилась, только при новом нажатии *Ctrl-F9*. И так далее.

Убирается точка прерывания точно так же, как ставится - *Ctrl-F8*.

“Иди до курсора” - Goto cursor. Это еще один способ остановки в нужной вам строке. Поставьте курсор в нужную строку и нажмите *F4*. Программа будет выполняться до тех пор, пока не наткнется на строку с курсором. А теперь поставьте курсор в другую строку и снова нажмите *F4*. Программа продолжит работу с того места, где остановилась, и будет выполняться до тех пор, пока не наткнется на строку с курсором. И так далее.

Кстати, в процессе выполнения программы вы можете достаточно свободно переключаться между разными способами ее выполнения – *F7*, *F4* или *Ctrl-F9*.

Испольование Write и ReadLn. Если программа работает с графикой, то пользоваться вышеописанными средствами отладки или неудобно или нельзя. Не беда. Еще до существования всех и всяческих отладчиков программисты пользовались вспомогательной печатью значений переменных. Действительно, вместо использования окна отладчика для подсматривания за переменными *n* и *k*, вы просто можете в программу вставить дополнительный оператор *Write(' n=',n,' k=',k)*. А вместо пошагового режима можете в нужных местах программы поставить *ReadLn*. Когда программа будет отлажена, вы уберете эти дополнительные операторы.

Только имейте в виду, что когда экран с вашими картинками донизу будет заполнен информацией, напечатанной оператором *Write*, он начнет прокручиваться вверх, а вместе с ним вверх уползут и ваши картинки. Чтобы избежать этого, нужно сделать так, чтобы не допечатав донизу, компьютер при помощи *ReadLn* сделал паузу (чтобы вы успели посмотреть результаты печати) и снова стал печатать бы сверху. А для печати сверху необходимо вставить в нужное место программы оператор *GotoXY* (см.15.7). А уж для этого вам нужно использовать модуль CRT (см. 9.1) и вставить в начало программы оператор *DirectVideo:=false* (см.10.3).

Работа с несколькими окнами.

Выполните, пожалуйста, **File→New**. Еще раз **File→New** и еще. У вас на экране появился каскад из нескольких окон. В каждом из них вы можете писать независимую программу, независимо выполнять ее и сохранять на диске. Листаем окна клавишей **F6**.

В обычной практике программиста несколько окон появляются на экране не в результате **File→New**, а в результате **File→Open**, когда программисту нужно или сравнить работу нескольких старых программ или скопировать часть текста из одной программы в другую.

То окно, в котором вы щелкнули мышкой, становится **активным**. Его легко отличить по двойной рамочке и по тому, что оно ближе к вам, чем остальные окна, и загораживает их. Остальные окна имеют одинарную рамочку. Когда вы запускаете программу на выполнение, то выполняться будет всегда программа из активного окна

(Если только у вас не указан какой-нибудь файл в опции **Compile→Primary file**. Если указан, то выберите **Compile→Clear primary file**).

Чтобы «достать» окна, загороженные другими, используйте мышку или **F6**.

Действия с одним окном:

- Каждое окно можно таскать по экрану, ухватившись мышкой за заголовок (это значит – поставить мышку на верхнюю грань окна, где записан заголовок, нажать левую кнопку мыши и не отпуская ее, таскать мышь по экрану).
- Можно изменять размеры окна, таская его правый нижний угол.
- Можно распахивать окно на весь экран и снова уменьшать клавишей **F5** или щелчком мыши по стрелке в правом верхнем углу окна.
- Можно уничтожить окно клавишами **Alt-F3** или щелчком мыши по кнопке в левом верхнем углу окна.
- Прокручивать текст в окне можно при помощи полос прокрутки на правой и нижней грани окна.

У начинающего программиста наличие нескольких окон на экране создает впечатление беспорядка. Не поддавайтесь соблазну уничтожать «лишние» окна. Они наверняка вам понадобятся чуть позже, а наводить порядок вы со временем научитесь.

Копирование и перемещение фрагментов текста

Часто в программах попадаются одинаковые или почти одинаковые фрагменты. Чем переписывать их заново, быстрее скопировать.

Поставьте мышку на первую букву копируемого фрагмента, нажмите левую клавишу мыши и, не отпуская ее, ведите мышь на конец копируемого фрагмента. Фрагмент будет выделен подсветкой. Если в процессе ваша рука дрогнет, то на экране будут происходить страшные вещи. Сохраняйте хладнокровие и ни за что не отпускайте клавишу мыши. Когда вы доведете ее до последней буквы фрагмента, все уляжется. Можете отпускать мышь. Выделенный же фрагмент Паскаль умеет перемещать и копировать в другое место. Следующим образом:

Перемещение: Дайте команду **Edit→Cut** - фрагмент исчез. Паскаль запомнил его в специальном месте памяти, которое называется **буфер обмена** или временный буфер. Теперь поставьте мигающий курсор текстового редактора в нужное место (для этого достаточно щелкнуть там мышкой). Теперь **Edit→Paste** - то, что было в буфере обмена, переносится на место курсора.

Копирование: Дайте команду **Edit→Copy** – фрагмент не исчез. Остальное - аналогично перемещению.

Если у вас на экране несколько окон, то вы точно так же можете копировать фрагменты из одного окна в другое.

Обзор популярных команд меню

Часто бывает так, что свои программы вы храните в одном каталоге, а при нажатии *File→Open* или *File→Save* Паскаль предлагает совсем другой каталог, за тридевять земель от нужного, что неудобно. **File→Change dir…** дает вам возможность самому задать предлагаемый каталог. Делайте двойные щелчки по именам каталогов на дереве каталогов, пока не доберетесь до нужного.

Если вы при вводе программы сделали какую-то глупость и в окне редактора ВСЕ ОЧЕНЬ ПЛОХО!!! – не паникуйте – ничего не пропало! **Edit→Undo** означает последовательную отмену ваших действий. А **Edit→Redo** означает последовательный возврат ваших отмененных действий.

Установите флажок (крестик) в **Options→Compiler…→Compiler Options→Runtime Errors →Range Checking** для того, чтобы Паскаль во время выполнения программы проверял, не вышли ли значения переменных и индексы, используемые в вашей программе, за указанные в разделе описаний пределы. Одно замечание: режим проверки включается не в тот момент, когда вы установили флажок, а тогда, когда после установки флажка вы компилируете программу. А если она была уже раньше откомпилирована и больше не хочет (так как не изменялась с тех пор)? Тогда измените ее как-нибудь – например, вставьте куда-нибудь ничего не значащий пробел и запустите на выполнение. Теперь ей придется перед выполнением подвергнуться компиляции.

Если при запуске программы, обращающейся к стандартным модулям, Паскаль жалуется, что не может найти модуль (*File not found (GRAPH.TPU)*), то проверьте, что записано в **Options → Directories… → Unit directories**. Там должна быть указана дорожка к файлам стандартных модулей и в том числе – к модулю *GRAPH.TPU*. Они обычно размещены в каталоге *Units* главного каталога Паскаля.

Если вы недовольны тем, что на экране умещается только 25 строк программного текста, воспользуйтесь **Options→Environment→Preferences→Screen Sizes**

Если вам не нравятся цвета, предлагаемые средой Паскаля, то воспользуйтесь **Options → Environment → Colors**. Правда, для этого нужно знать английские термины для элементов оформления этой среды.

Если у вас на экране мешанина окон и вы хотите навести среди них порядок, воспользуйтесь **Window → Cascade** или **Window → Tile**. Список имен всех окон на экране вы найдете в **Window → List…**.

Если вы знаете английский язык, то можете воспользоваться системой помощи **Help**. Если вы хотите узнать поподробнее о каком-нибудь операторе или другом слове из своей программы, являющимся стандартным для Паскаля, поставьте на него мигающий курсор и нажмите *Ctrl-F1*. В открывшемся окне вы можете найти ссылку **Sample Code**, что означает пример программы с использованием интересующего вас слова. Вы можете этот пример скопировать оттуда обычным образом в окно редактора и выполнить его.

Создание исполнимых файлов (exe)

Ваши паскалевские программы могут работать только «из-под» Паскаля. Это значит, что если вы запишете свою любимую программу на компьютер, где Паскаля нет, то там вы ее запустить не сможете. Чтобы все-таки быть запущенной и без Паскаля, программа должна быть вами заранее откомпилирована и записана на диск с расширением **exe** (то есть стать «экзешным» файлом или, как еще говорят, – «экзешником»).[11]

Сделать это просто: **Compile** - затем убедитесь, что в выпавшем меню опция **Destination** (*куда* записать откомпилированную программу) находится в положении **Disk**, а не **Memory** (оперативная память). Если это не так, то щелкните по этой опции. Теперь все в порядке. Снова **Compile → Compile** и все - ваш файл с расширением *exe* находится на диске, в том же каталоге и с тем же именем, что и исходный файл с расширением *pas*.

Если ваша программа использует модуль *Graph*, то одного «экзешного» файла недостаточно. Для того, чтобы он запустился на чужом компьютере, там, в одном каталоге с ним, должны находиться файл графического драйвера **egavga.bgi** (или **cga.bgi**, если у вас видеоадаптер CGA) и все используемые вами векторные шрифты (файлы с расширением **chr**). Скопируйте их из каталога *BGI* главного каталога Паскаля.

[11] Я приношу свои извинения за жаргон. Но что делать, если он широко распространен среди программистов. В конце концов, многие общепринятые слова раньше были жаргоном.

Приложения. Справочный материал

П1. Как вводить программу в компьютер или работа с текстом в текстовом редакторе

Ввод вашей программы в компьютер производится при помощи специальной программы, которая называется текстовым редактором и входит в состав Паскаля. Паскалевская программа - это обычный текст, поэтому нам достаточно освоить работу с текстом в текстовом редакторе.

Работа с одной строкой текста

Когда мы начинаем работать в текстовом редакторе, перед нами обычно - пустой экран, по краям которого могут располагаться различные меню и "письменные принадлежности". На них мы внимания пока не обращаем. Пустое пространство экрана - наш лист, и на нем мы будем писать.

Я бы не хотел, чтобы мы учились вводить сразу же текст программы. Эффективнее учиться на специально подобранных маленьких текстах. А чтобы знать, как реагировать на неожиданности, нужно перед тем, как начать нажимать клавиши, проглядеть не очень внимательно дальнейший материал вплоть до «Работы с несколькими строками».

Ввод строки. Пусть мы хотим ввести строчку "*юный пионер Коля любит Bubble Gum*". Первая клавиша, по которой мы должны щелкнуть, - русская буква "*ю*". Но где она появится на экране? - В том месте, где сейчас мигает курсор (см.4.4). А в начале работы редактора он должен мигать в левом верхнем углу.

Щелкаем по клавише "*ю*" - и на экране на месте курсора возникает буква "*ю*", а сам курсор перемещается чуть вправо, там потом возникнет следующая буква. (Если получилась не буква "*ю*", а точка, значит прочитайте чуть ниже про английские и русские буквы. Если буква получилась заглавной, почитайте про заглавные буквы.) Вот какая получается картина - "*ю_*". По клавише именно щелкаем, «клюем» ее, а не нажимаем, потому что компьютер настроен так, что если мы задерживаем палец на клавише дольше некоторой доли секунды, он считает, что нажатий было не одно, а два, еще подольше – три, и так далее, а это значит, что на экране мы увидим "*ююююююююю_*". Поэтому в дальнейшем изложении, когда я говорю «Нажмите клавишу», я буду иметь в виду «Щелкните по клавише» за исключением специально оговоренных случаев.

Получив на экране "*ю*", щелкнем по "*н*". На экране видим "*юн_*". И так далее, пока на экране мы не получим "*юный_*".

Если мы в процессе работы случайно нажали на клавишу не с той буквой, то щелкнем по клавише *BackSpace*. Она стирает последнюю введенную букву. Эта клавиша имеет еще маркировку *BS* или ⇦.

После ввода слова "*юный*" нужно ввести пробел перед следующим словом. Для этого щелкните по самой длинной горизонтальной клавише. Затем аналогично вводите слово "*пионер*", пробел и так далее до слова "*Коля*".

Заглавные буквы. Чтобы буква "*к*" в слове "*Коля*" получилась заглавной, нужно нажать на клавишу *Shift* и держать ее нажатой. Смело держите ее нажатой сколько угодно - ничего плохого от этого не произойдет. Затем, удерживая ее нажатой, щелкните по букве "*к*" - она получится заглавной. Теперь отпустите *Shift*. Можно работать дальше со строчными буквами. Иногда клавиша *Shift* имеет маркировку ⇧.

Ненароком в процессе работы вы можете нажать клавишу *CapsLock* и не заметить этого. После этого при нажатии на любую буквенную клавишу вы получите заглавную букву вместо строчной. При этом в правом верхнем углу клавиатуры горит индикатор *CapsLock*. Еще раз нажмите на *CapsLock* и все вернется на свои места.

Английские и русские буквы. Вот вы дошли до английского слова "*Bubble Gum*". Как его набрать? Вы уже заметили, что обычно на большинстве клавиш букв две - сверху английская, снизу русская. Предположим, до сих пор у вас при нажатии на клавишу всегда выходила русская буква. Говорят, что вы работали в **русском регистре**. Если вы нажмете определенную специальную клавишу или пару специальных клавиш, то теперь при нажатии на любую буквенную клавишу у вас будут получаться английские буквы, пока вы снова не нажмете на эти самые или, возможно, на другие специальные клавиши, чтобы вернуться к русским буквам.

Управляет процессом ввода информации с клавиатуры специальная программа, называемая **драйвером клавиатуры**. Драйверы бывают разные и от них зависит, какие именно специальные клавиши нужно нажимать для переключения из русского регистра в английский и обратно. Если вы предварительно не узнали, какие это клавиши, то попробуйте такие сочетания:

- *Shift + Shift* На клавиатуре имеется две клавиши *Shift*. Имеется в виду, что удерживая нажатой одну клавишу *Shift* вам нужно щелкнуть по другой.
- *Ctrl* Клавиш *Ctrl* тоже две. Обычно имеется в виду правая. Иногда правая переключает в один язык, левая – в другой.
- *Ctrl + Shift* Попробуйте разные сочетания. Иногда правая пара клавиш переводит клавиатуру в английский регистр, а левая – в русский.
- *Alt + Shift* Попробуйте разные сочетания.
- Если ничего из вышеназванного не подойдет, попробуйте любые сочетания клавиш *Ctrl, Alt, Shift*.
- Если и это не поможет, то попробуйте узнать, а запущен ли у вас драйвер клавиатуры.

Знаки препинания. Вы набрали все предложение. Теперь вам самое время поставить точку, а мне поговорить о знаках препинания. В отличие от букв, их на клавиатуре не сразу и найдешь. Над буквами вы увидите горизонтальный ряд клавиш с цифрами от 0 до 9. На эти же клавиши нанесены и многие знаки препинания. Вы их сможете получить при нажатой клавише *Shift*. Однако многие драйверы клавиатуры получают их совсем не там, где они нанесены краской на клавиши. Например, точка обычно живет на клавише "*?*" рядом с правой клавишей *Shift*, а запятая - на ней же с нажатой *Shift*. Попозже поэкспериментируйте. Понажимайте все подозрительные клавиши просто так или при нажатой *Shift*. Но не сейчас. Сейчас нам нужно идти дальше.

Удаление букв из текста. Вот вы напечатали всю строку "*юный пионер Коля любит Bubble Gum*". Теперь попробуем удалить слово "*пионер*". Для этого нам нужно уметь перемещать курсор. Для перемещения курсора служат четыре клавиши внизу клавиатуры: ← → ↑ ↓. В нашем случае будут работать только первые две. Попробуйте, как они работают. Затем поставьте курсор на пробел между словом "*пионер*" и словом "*Коля*". Мы уже начинаем привыкать, что если нажать какую-нибудь клавишу, то что-то произойдет именно в том месте, где курсор. Чтобы стереть по очереди все буквы слова "*пионер*", несколько раз нажмите на клавишу *BackSpace*. Обратите внимание, что буквы слова "*пионер*" слева от курсора исчезают по одной, текст справа от курсора смыкается налево, так что пустого пространства на месте слова "*пионер*" не остается. У вас должно получиться "*юный Коля любит Bubble Gum*".

Для стирания символов существует еще одна клавиша - *Delete*. Иногда она имеет маркировку *Del*. Давайте сотрем слово "*любит*". Поставим курсор на пробел между словом "*Коля*" и словом "*любит*". Нажмем несколько раз на *Delete*. Слово любит "стерлось", а текст справа от курсора снова сомкнулся налево.

Таким образом, клавиша *BackSpace* стирает символ слева от курсора а клавиша *Delete* – тот, на котором стоит курсор. В обоих случаях текст справа от курсора смыкается налево на место курсора.

Вставка букв в текст. Теперь у нас на экране строка "*юный Коля Bubble Gum*". Давайте вставим перед словом "*Коля*" слово "*бойскаут*". Для этого поставим курсор на букву "*К*" в слове "*Коля*". После этого напечатаем слово "*бойскаут*" и пробел. Мы увидим, что буквы слова "*бойскаут*" появляются на месте курсора, "*Коля*" вместе с "*Bubble Gum*" ушли направо и мы достигли поставленной цели. Теперь у нас на экране строка "*юный бойскаут Коля Bubble Gum*".

Этот способ работы текстового редактора, когда вставляемый текст отодвигает вправо остальной текст, называется **режимом вставки**. Если вы нажмете на клавишу *Insert*, иногда маркируемую *Ins*, то перейдете в **режим замещения**, когда текст не будет отодвигаться, а "*бойскаут*" сотрет "*Колю*". Чтобы вернуться в режим вставки, еще раз нажмите на *Insert*.

Работа с несколькими строками

Ваша задача – ввести такой текст из нескольких строк:

В небе

Облака из серой ваты
Сыровато-сероваты,
Не беда - ведь я привык.

В луже
Эта вата намокает
И волнуясь пробегает
Под водою мой двойник.

Чтобы знать, как реагировать на неожиданности, нужно перед тем, как начать нажимать клавиши, проглядеть не очень внимательно дальнейший материал вплоть до конца приложения П1.

Ввод нескольких строк. Как сделать так, чтобы, введя слова «в небе», следующие слова начать с новой строки? Для этого нужно нажать клавишу *Enter*, по-другому *Return*, по-другому «Клавиша ввода». Курсор перепрыгнет в начало следующей строки. Введя вторую строку, снова нажмите на клавишу ввода и так далее. Нажатие клавиши ввода - единственный способ в текстовом редакторе Паскаля начать следующую строку. Клавиша перемещения курсора ↓ здесь не поможет.

А теперь введите все восемь строк задания.

Перемещение курсора по экрану. При помощи четырех клавиш перемещения курсора ← → ↑ ↓ потренируйтесь перемещать курсор куда только можно. Вы скоро обнаружите, что курсор можно свободно перемещать только там, где имеется текст. Ни правее, ни ниже введенного текста курсор переместить не удается. Поначалу вам это может показаться непривычно и неприятно, и вы захотите расширить поле действия курсора. Удовлетворить вашу прихоть довольно легко.

Введенным текстом считаются не только буквы, но и пробелы. Доведите курсор до правого края строки и нажмите несколько раз на пробел. У вас в этой строке образовалось пустое поле, по которому свободно может ходить курсор. Так вы можете сделать во всех строках.

А если вы хотите переместить курсор ниже текста? Подведя курсор в правый край самой нижней строки, нажмите на клавишу ввода несколько раз. У вас ниже текста образовалось несколько пустых строк, по которым вверх-вниз может свободно ходить курсор. Все эти строки вы можете, если уж вам так хочется, заполнить пробелами. Вот у вас и получится ниже текста спортплощадка для курсора.

Я назвал это прихотью, так как при вводе текста это никогда не бывает нужно. Но то, что вы сейчас проделали, вам полезно для свободной ориентации на листе.

Собственно работа с несколькими строками.

А теперь вам полезно выполнить несколько заданий.

Чтобы вставить пустые строки между строчкой «*Не беда - ведь я привык.*» и строчкой «*В луже*», поставьте курсор в конец первой из этих строк или в начало второй и несколько раз нажмите клавишу ввода.

А как теперь убрать эти пустые строки? Поставьте курсор в начало самой верхней из пустых строк и несколько раз нажмите *Delete*.

Как разделить строку на две части? Например, вместо «*Не беда - ведь я привык.*» нужно получить

Не беда –
ведь я привык.

Поставьте курсор на букву "*в*" и нажмите клавишу ввода.

А как слить эти две строки? Поставьте курсор в правый конец верхней из этих строк и нажмите *Delete* один или несколько раз, пока строки не сольются.

Невидимые символы. Все эти правила могут показаться запутанными и не имеющими внутренней логики. А логика есть. И если вы ее поймете, то и правил запоминать не нужно. Вот она:

Нажатие на клавишу ввода вызывает появление на экране в том месте, где был перед нажатием курсор, специального невидимого символа, точно так же, как нажатие на клавишу пробела вызывает появление невидимого символа - пустого места. Обозначим для удобства символ клавиши ввода - π.

Рассмотрим с новой точки зрения действие различных клавиш:

- Нажатие на любую буквенную клавишу или пробел вызывает вставку в текст на место курсора соответствующей буквы или пробела, а вся правая часть текста сдвигается вправо.
- Нажатие на клавишу ввода вызывает перемещение вниз на одну строку всего текста, начиная от курсора правее и ниже, причем правая часть текста в строке, где был курсор, перемещается не только вниз, но и в начало следующей строки. Отсюда видно, что в текстовом редакторе Паскаля строка кончается обязательно символом π. Это не относится к тем текстовым редакторам, которые переводят строку автоматически. Кроме как в конце строки, символ π нигде встречаться не может.

- Клавиша *Delete* стирает любой символ, на котором стоит курсор, будь то буква, пробел или **π**. Стирание символа уничтожает не только сам символ, но и его действие. Поэтому, стерев **π**, мы выполняем действие, обратное действию клавиши ввода, то есть нижние строки поднимаются, а ближайшая нижняя сливается с текущей.
- Аналогично действует клавиша *BackSpace*.

Экран - окно на лист с текстом. Когда вы вводите большой текст, то в конце концов доходите до нижнего края экрана. Продолжайте работать как ни в чем не бывало. Нажав в очередной раз на клавишу ввода, вы обнаруживаете, что весь текст на экране ушел немного вверх. Оказывается, экран - это не лист с текстом, а всего лишь окно на этот большой лист. Упираясь курсором в край экрана, вы можете двигаться в нужном направлении. В паскалевском текстовом редакторе то же самое происходит, когда слишком далеко продолжаешь строку вправо - текст уходит влево.

Впечатление такое, что имеется большой неподвижный лист с текстом, а экран монитора является небольшим подвижным окном, через которое вы можете видеть этот лист. Движением окна можно управлять с клавиатуры или мышкой щелкая по полосам прокрутки.

П2. Файловая система магнитного диска

Здесь мы разберем структуру хранения информации на жестких дисках и дискетах. Магнитный диск - вместительное хранилище, на котором вы храните самые разные объекты: и программы, и тексты, и картинки, и музыку, и даже небольшие видеофильмы. Каждый из этих объектов, хранящихся на диске, принято называть **файлом**.

У каждого файла должно быть **имя,** например, *Sveta25*. Файлы появляются на диске двумя способами: или вы их переписываете на диск из другого места и тогда у них уже есть имя, или вы сами создаете файл и тогда имя ему придумываете.

Обычно на жестком диске хранятся сотни и тысячи файлов. Просмотр содержимого диска - операция, которую вы выполняете очень часто, и поэтому многие программы, включая Паскаль, позволяют вам это делать. При этом вы всегда видите на мониторе список имен файлов.

Обычно вам нужно найти среди них какой-то один, чтобы начать с ним работать. Но когда в списке тысяча файлов, найти среди них что-то довольно затруднительно. Нужно навести среди файлов какой-то порядок, какую-то систему.

Вспомните, как вы храните свои вещи в шкафу. На одной полке у вас книги, на другой – тетради и т.д. Но этого мало. Полка с книгами у вас разделяется на отделения: в одном – учебники, в другом – фантастика. Если нужно, то некоторые отделения можно разбить помельче, например, фантастика про космос, фантастика про роботов и т.д.

Для чего вам все эти отделения? Для того, чтобы легче было искать нужный предмет. На диске принята та же система. Только все эти полки и отделения называются одним термином – **каталог**. В современных компьютерах, работающих под управлением операционных систем (ОС) Windows 95, Windows 98 и Windows NT, вместо термина «каталог» употребляется термин **папка**, а многие файлы называются **документами**.

Каталоги, как и файлы, вы или создаете на диске сами или переписываете откуда-нибудь. Каждый каталог тоже имеет имя.

Имена файлов и каталогов

Имена файлам и каталогам можно придумывать произвольные, но с некоторыми ограничениями. Ограничения зависят прежде всего от ОС, управляющей компьютером. ОС MS-DOS – самая строгая. Она требует, чтобы имя файла или каталога было не длиннее 8 символов и состояло из латинских букв, цифр, еще кое-каких символов и не содержало пробелов. Windows 95 и Windows NT добрее – имя может быть длиной до 255 символов и содержать пробелы. А если две эти ОС русифицированы, то есть приспособлены для России, то имена могут быть и русскими.

Сегодня в России используются все три упомянутые ОС. Если ваш компьютер работает под управлением русифицированой Windows 95 и вы назвали файл *Это фото Рыжика*, то с вашим файлом могут быть проблемы на компьютере, работающем под управлением нерусифицированой Windows 95 и тем более MS-DOS. А вот если бы вы назвали файл *Red_Cat*, то проблем бы не было.

Ограничения зависят также и от самой программы. Многие программы (в том числе Турбо-Паскаль 7.0) были созданы для работы под управлением MS-DOS и поэтому строги к именам, даже работая под Windows 95 или Windows NT.

Имя многих файлов имеет справа добавку, состоящую из точки и справа от нее одного, двух или трех символов. Эта «фамилия» называется **расширением**. Например, файл, в котором вы описываете, как Ира печет булки, вы могли бы назвать *Bulki.Ira*.

Расширение может автоматически и незаметно для вас приписываться к имени файла программой, в которой вы работаете. Так, если вы в Паскале написали программу и решили записать ее на диск под именем *Train*, то на самом деле файл этой программы на диске будет иметь имя *Train.pas*. По расширению программа узнает

«свои» файлы, а опытные пользователи узнают, в какой программе файл был создан. И наконец, расширения у файла может и не быть.

Пример структуры каталогов на диске

Пусть вы решили на одной из своих дискет создать хранилище информации, а именно:

- *Игры*
- *Ваши программы на Паскале*
- *Переписка с друзьями*

Для этого на пустой дискете вы создали три каталога:

- *Games*
- *Pascal*
- *Letters*

В каталоге *Games* у вас будут храниться файлы, представляющие собой программы игр: *tetris.exe*, *arcanoid.com*, *cat.exe*.

Программы на Паскале вы решили разделить на три категории:

- *Программы для рисования*
- *Музыкальные программы*
- *Все остальные*

Для этого внутри каталога *Pascal* вы создаете три каталога:

- *Graph*
- *Music*
- *Other*

Внутри каталога *Graph* у вас могут находиться, например, такие файлы: *train.pas*, *ball.pas*.

У вас есть два друга, Игорь и Эдик. Вы пишете друг другу письма, созданные в текстовых редакторах ваших компьютеров. Свою переписку вы храните на дискетах. Поэтому внутри каталога *Letters* вы создаете два каталога:

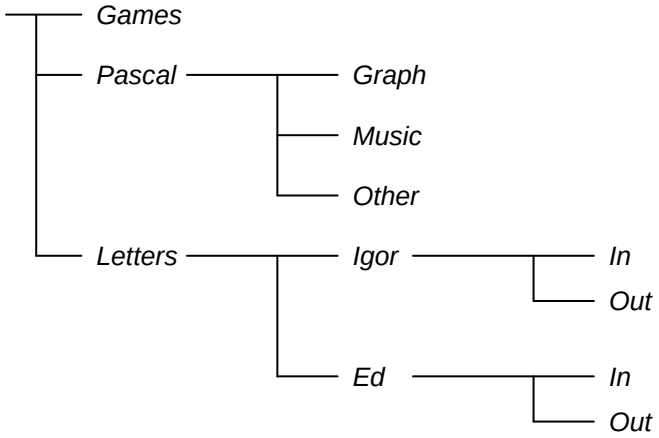
- *Igor* (для переписки с Игорем)
- *Ed* (для переписки с Эдиком)

Чтобы отличить письма от Игоря от писем к Игорю, вы создаете внутри каталога *Igor* два каталога:

- *In* (для писем от Игоря)
- *Out*(для писем к Игорю)

Такие же два каталога вы создаете внутри каталога *Ed*. Внутри каталога *In* каталога *Ed* могут находиться, например, файлы *25may97.txt* и *03june97.txt*.

Вот структура каталогов на вашей дискете, представленная наглядно в виде **дерева**:



Действительно, эта картинка похожа на лежащее дерево. Слева корень, справа самые тонкие веточки. Листья (файлы) на дереве обычно не изображают, так как их слишком много.

Самый левый каталог (в нем находятся каталоги *Games, Pascal, Letters*, а сам он ни в какие каталоги не входит) называется **корневым каталогом** диска или **корнем**. Если каталог (например, *Music*) входит внутрь другого каталога (*Pascal*), то он называется **подкаталогом** этого каталога. В этом случае *Pascal* называют также его **родительским каталогом**.

15.1.1. Логические диски. Путь (дорожка) к файлу

Многие программы позволяют вам создавать, удалять и переименовывать файлы и каталоги, копировать и переносить их из любого каталога в любой другой и обратно, с жесткого диска на дискету и обратно. Самые известные из таких программ – Windows и Norton Commander.

В процессе общения с этими программами вам приходится объяснять им, где, в каком каталоге находится такой-то файл или каталог, и самим понимать их объяснения. Например, вам нужно понимать, что значит запись *B:\Letters\Ed\In\25may97.txt*. Для этого сначала разберем, что такое логические диски.

Пусть на вашем компьютере есть дисковод для трехдюймовых дисков, дисковод для пятидюймовых дисков и дисковод для компакт-дисков. Компьютер именует все эти дисководы буквами латинского алфавита. Дисководы для дискет должны иметь имена *A* и *B*. Пусть у вас пятидюймовый дисковод имеет имя *A*, трехдюймовый - *B*. Жесткий диск почти всегда имеет имя *C*. Однако, у многих жестких дисков имеется странность, доставшаяся им, как аппендицит, от старых версий ОС. Эта странность состоит в том, что винчестер делится на несколько независимых участков. Каждый участок называется логическим диском. Эти логические диски получают имена *C, D, E* и т. д. ОС предлагает нам пользоваться этими логическими дисками, как независимыми винчестерами. Что ж, в принципе, нам все равно, пользователь может даже и не знать, что у него на компьютере не несколько жестких дисков, а один. Компакт-диск тоже получает одну из букв.

Итак, как же понимать запись *B:\Letters\Ed\In\25may97.txt*? Она означает, что файл с именем *25may97.txt* находится в каталоге *In*, который находится в каталоге *Ed*, который находится в каталоге *Letters*, который находится в корневом каталоге дискеты, вставленной в трехдюймовый дисковод. Эта запись называется **путем** или **дорожкой** к файлу *25may97.txt*.

Эта запись довольно длинная и скучная. Ведь не пишете же вы на письме такой адрес: Планета Земля, Россия, г.Пенза, ул.Паскаля, д.1, кв.3, Попову А.А. Довольно часто компьютер «чувствует», что вы работаете в некотором каталоге, и в этом случае вам достаточно указать ему только имя файла.

П3. Список некоторых операций, процедур и функций Паскаля

Приведу список большинства пройденных нами операций и стандартных процедур и функций с указанием типа их параметров (или по-другому – аргументов и результата)

Математика

Sqr (A)	аргумент A – любого числового типа, результат - того же типа, что и A
Sqrt (A)	аргумент A – любого числового типа, результат - того же типа, что и A
Pi	значение - вещественное
Frac (A)	аргумент A и результат - вещественные
Int (A)	аргумент A и результат - вещественные
Round (A)	аргумент A вещественный, результат - LongInt
Abs (A)	аргумент A – любого числового типа, результат - того же типа, что и A
Random	результат - вещественный
Random (A)	аргумент A и результат - Word
Randomize	процедура
A div B	операнды A и B и результат операции div - только целочисленного типа
A mod B	операнды A и B и результат операции mod - только целочисленного типа

Модуль CRT

Процедура или функция	**Смысл**
Sound (A :Word)	звук
NoSound	нет звука
Delay (A :Word)	отсрочка
ReadKey :Char	читай клавишу
KeyPressed :Boolean	клавиша нажата
TextColor (A :Byte)	цвет текста
TextBackground (A :Byte)	цвет фона
ClrScr	очистка экрана
GotoXY (X,Y :Byte)	иди к икс игрек

Модуль Graph

InitGraph (d,m :Integer; path :String)	инициализация графики
CloseGraph	закрытие графики
PutPixel (x,y :Integer; color :Word)	поставь пиксел
GetPixel (x,y :Integer) :Word	получи пиксел
Line(x1,y1, x2,y2 :Integer)	отрезок прямой линии
Rectangle (x1,y1, x2,y2 :Integer)	прямоугольник
Bar (x1,y1, x2,y2:Integer)	залитый прямоугольник
Bar3D (x1,y1, x2,y2:Integer; tol:Word; top:Boolean)	параллелепипед
Circle (x,y :Integer; r :Word)	окружность

Arc (x,y :Integer; fi1,fi2,r :Word)	дуга окружности
PieSlice (x,y :Integer; fi1,fi2,r :Word)	кусок пирога (залитый сектор круга)
Ellipse (x,y :Integer; fi1,fi2, rx,ry :Word)	эллипс
Sector (x,y :Integer; fi1,fi2, rx,ry :Word)	залитый сектор эллипса
FillEllipse (x,y :Integer; rx,ry :Word)	залитый эллипс
SetColor (color :Word)	установить цвет линий
SetLineStile (ls,uzor,tolsh :Word)	установить стиль линий
SetFillStyle(uzor,color :Word)	установить стиль заливки
FloodFill(x,y :Integer; granitsa :Word)	заливка
GetMaxX :Integer	получи максимальный икс
GetMaxY :Integer	получи максимальный игрек
GetImage (x1,y1, x2,y2 :Integer; var A[12])	получи изображение
PutImage (x1,y1 :Integer; var A; mode :Word)	помести изображение
ImageSize (x1,y1, x2,y2 :Integer)	размер изображения
OutTextXY (x,y :Integer; stroka :String)	вывод текста
SetTextStyle (shrift,napravl,razmer :Word)	установи стиль текста
SetUserCharSize (A,B, C,D :Word)	установи пользовательский размер символов

Модуль DOS

GetTime (var chas,min,sec,sotki :Word)	получи время
SetTime (chas,min,sec,sotki :Word)	установи время
GetDate (var god,mes,den,dennedeli :Word)	получи дату
SetDate (god,mes,den :Word)	установи дату

Работа со строками

Pos (s1,s :String) :Byte	позиция
Length (s :String) :Byte	длина
Copy (s :String; a,b :Integer) :String	копируй
Delete (var s :String; a,b :Integer)	удаляй
Insert (var s :String; s1 :String; a :Integer)	вставляй

Работа с файлами

Assign (var f: Text; filename :String)	*упрощенно*	присвоить
Rewrite (var f: Text)	*упрощенно*	переписать
Reset (var f: Text)	*упрощенно*	переустановить
Close (var f: Text)	*упрощенно*	закрыть
Append (var f: Text)		добавить

Прочие

Chr (A :Byte) :Char	символ
GetMem(P :Pointer; razmer :Word)	выдели память
New (var p: Pointer)	новый

П4. Произношение английских слов

Здесь приведена транскрипция всех английских слов, встречающихся в тексте, кроме самых простых, таких как *in*. Кроме транскрипции произношение английских слов мне пришлось привести еще и русскими буквами, во-первых потому, что не все разбираются в значках транскрипции, а во-вторых потому, что многие аббревиатуры в среде русскоязычных программистов принято произносить на латинском или на смеси английского с латинским. Значок ударения в русском произношении я по техническим причинам ставил не над гласной буквой, как принято в России, а перед ударным слогом.

[12] Здесь и в следующей процедуре **A** – так называемый **бестиповой** параметр-переменная, который мы с вами не проходили.

Слово	Транс-крипция	Обычное произношение
Add watch		Эд’воч
Alt		Альт
AND		Энд
Append		Э’пенд
array		Э’рэй
Assign		Э’сайн
BackSpace		Бэк’спэйс
Bar3D		Бартри’дэ
Begin		Би’гин
Boolean		‘Булиэн
Byte		Байт
CapsLock		Капс’лок
case		Кэйс
Char		Кэр
Chr		Сиэйч’а
Circle		Сёкл
ClearDevice		Клиэди’вайс
Close		‘Клоуз
CloseGraph		Клоуз’граф
ClrScr		Клиэ’скрин
Compile		Ком’пайл
Copy		‘Копи
CRT		Цээр’тэ
Ctrl		‘Контрол
Cut		Кат
Debug		Ди’баг
Delay		Ди’лэй
Delete		Ди’лит
DirectVideo		Директ’видео
do		Ду
Double		Дабл
downto		‘Даунту
Edit		‘Эдит
Ellipse		Эллипс
else		Элз
End		Энд
EOF		Энд оф ‘файл
exe		‘Экзе
Exit		‘Эксит
Extended		Икс’тендед
false		Фолс
File		Файл
FillEllipse		Фил’эллипс
FloodFill		Флад’фил
for		Фо
Forward		‘Форвард
Frac		Фрак
GetDate		Гэт’дэйт
GetImage		Гэт’имэдж
GetMaxX		Гэтмакс’икс
GetMaxY		Гэтмакс’игрек
GetMem		Гэт’мэм
GetPixel		Гэт’пиксел
GetTime		Гэт’тайм
goto		‘Гоуту
Goto cursor		Гоуту’курсор
GotoXY		Гоутуикс’игрек
Graph		Граф
Halt		Хальт
if		Иф
ImageSize		‘Имэджсайз
IMPLEMENTATION		Имплемен’тэйшн
InitGraph		‘Инитграф
Insert		Ин’сёт
Integer		‘Интеджер
KeyPressed		Ки’пресд
LABEL		Лэйбл
Length		Ленгс
Line		Лайн
LongInt		Лонг’инт
New		Нью
NormWidth		Норм’видс
NoSound		Ноу’саунд
of		Эв
Open		‘Опен
OR		О
Output		Аутпут
OutTextXY		Ауттекстикс’игрек
Paste		Пэйст
PC Speaker		Писи’спикер
PieSlice		Пай’слайс
Pos		По’зишн
PROCEDURE		Про’сиджэ
PROGRAM		‘Програм
PutImage		Пут’имэдж
PutPixel		Пут’пиксел
Random		‘Рэндом
Randomize		Рэндо’майз
Read		Рид
ReadKey		Рид’ки
ReadLn		Рид’лайн
Real		‘Риэл
record		Рекорд
Rectangle		Рек’тангл
repeat		Ри’пит
Reset		Ри’сет
Rewrite		Ри’райт
Round		‘Раунд
Run		Ран
Save		Сэйв
Save as		Сэйв’эз
SetColor		Сет’колор
SetDate		Сет’дэйт
SetFillStyle		Сетфил’стайл
SetLineStyle		Сетлайн’стайл
SetTextStyle		Сеттекст’стайл
SetTime		Сет’тайм
SetUserCharSize		Сет юзеркэр’сайз
Shift		Шифт
ShortInt		Шорт’инт
Single		Сингл
Sound		‘Саунд
Sqr		‘Сквэар
Sqrt		Сквэа’рут
String		Стринг
Succ		Сэ’ксэссор
Text		Текст
TextBackgrownd		Текстбэк’граунд
TextColor		Текст’колор
then		Зэн
ThickWidth		Сик’видс
to		Ту
TopOff		Топ’оф
TopOn		Топ’он
Trace into		Трэйс’инту
true		Тру

TYPE		Тайп
UNIT		‘Юнит
until		Ан’тил
User Screen		Юзер’скрин
Uses		‘Юэеэ
Watch		Воч

while		Вайл
Word		Ворд
Write		Райт
WriteLn		Райт’лайн

П5. Решения заданий

Задание 1

Программа для робота	Пояснения для нас с вами
1. ПОВТОРИ 5 РАЗ ШАГ ВПЕРЕД	Робот идет по коридору до дверей
2. НАЛЕВО	Робот поворачивается лицом к дверям
3. ПОВТОРИ 3 РАЗА ШАГ ВПЕРЕД	Робот подходит к стулу
4. ВОЗЬМИ ПРЕДМЕТ	Робот берет стул
5. ПОВТОРИ 2 РАЗА НАЛЕВО	Робот поворачивается кругом
6. ПОВТОРИ 3 РАЗА ШАГ ВПЕРЕД	Робот выходит в коридор
7. НАПРАВО	Поворачивается направо
8. ПОВТОРИ 5 РАЗ ШАГ ВПЕРЕД	Идет по коридору до исходного положения
9. ОПУСТИ ПРЕДМЕТ	Робот ставит стул на новое место

Задание 2

1 2 5 + 1 6 = 1 4 1

Задание 3

1	9	9	2										М	ы		н	а	ч	и	н	а	е	м	!								
4	8																															
Ш	е	с	т	ь	ю		ш	е	с	т	ь				3	6	.			А	р	и	ф	м	е	т	и	к	а	:	3	0		

Задание 4 0

Задание 5

```
VAR  a,b : LongInt;
BEGIN
    a:= 9000000;
    b:= 1000;
    WriteLn (a-b);
    ReadLn
END.
```

Задание 6 Паскаль выдает сообщение об ошибке: *Constant out of range* (см. сообщения об ошибках в части IV).

Задание 7 Паскаль выдает результат 211 или другой неправильный.

Задание 8 1001

Задание 9 -100

Задание 10 15 -10

Задание 11 82

Задание 12

```
VAR  dlina_dvora,          dlina_doma,
     shirina_dvora,        shirina_doma,
     pl_dvora,             pl_doma,
     svobodn_pl_dvora,     dlina_zabora        :Integer;
BEGIN
    dlina_dvora:=50;       dlina_doma:=20;
    shirina_dvora:=30;     shirina_doma:=10;
```

```
    pl_dvora := dlina_dvora * shirina_dvora;
    pl_doma  := dlina_doma  * shirina_doma;
    svobodn_pl_dvora := pl_dvora - pl_doma;
    dlina_zabora := 2*(dlina_dvora+shirina_dvora)-(dlina_doma+shirina_doma);

    WriteLn (pl_doma,'    ', svobodn_pl_dvora,'    ', dlina_zabora);
    ReadLn
END.
```

Задание 13

```
VAR  r    :Integer;      {r-радиус окружности}
     l, s :Real;         {l-длина окружности, s-площадь круга}
BEGIN
    r := 800;
    l := 2 * pi * r;
    s := pi * r * r;
    WriteLn (l :15:5, '    ', s:15:5);
    ReadLn
END.
```

Задание 14

```
VAR  t1, t2,                    {t1-время на первом отрезке, t2-на втором}
     v1, v2,                    {v1-скорость на первом отрезке, v2-на втором}
     s1, s2          :Integer;  {s1-первый отрезок пути, s2-второй}
     sredn_skorost   :Real;
BEGIN
    t1:=3;     t2:=2;
    v1:=80;    v2:=90;
    s1:=v1*t1;                  {путь равен скорость умножить на время}
    s2:=v2*t2;
    sredn_skorost := (s1+s2)/(t1+t2);
    WriteLn (sredn_skorost :10:3);
    ReadLn
END.
```

Задание 15

```
VAR  a,b,c, perimetr :Integer;   {a,b,c - стороны треугольника}
BEGIN
    a:=20;
    WriteLn ('Введите длины двух сторон треугольника');
    ReadLn(b,c);
    perimetr := a+b+c;           {периметр - это сумма сторон}
    WriteLn ('Периметр треугольника равен ' , perimetr);
    ReadLn
END.
```

Задание 16

```
VAR  t, v, s    :Real;       {t-время, v-скорость, s-путь}
BEGIN
    WriteLn ('Введите путь в метрах и скорость в м/с');
    ReadLn(s,v);
    t:=s/v;
    WriteLn ('Время = ', t :6:3, ' сек');
    ReadLn
END.
```

Задание 17

```
VAR  r1, r2,                    {r1-радиус орбиты первой планеты, r2-второй}
     v1, v2,                    {v1-скорость первой планеты, v2-второй}
     t1, t2          :Real;     {t1-продолжительность года первой планеты, t2-второй}
     nazvanie1, nazvanie2       :String;
BEGIN
   WriteLn('Введите название первой планеты');
   ReadLn(nazvanie1);
   WriteLn('Введите радиус орбиты и скорость первой планеты');
   ReadLn(r1,v1);
   WriteLn('Введите название второй планеты');
   ReadLn(nazvanie2);
   WriteLn('Введите радиус орбиты и скорость второй планеты');
   ReadLn(r2,v2);
   t1 := 2*pi*r1/v1;  {время = длина орбиты/скорость, а длина}
   t2 := 2*pi*r2/v2;  {орбиты равна два пи * радиус}
   WriteLn ('Продолжительность года на планете ', nazvanie1, ' - ', t1: 3:0,
            ' сут., а на планете ', nazvanie2, ' - ', t2 :3:0, ' сут.');
   ReadLn
END.
```

Задание 18 8

Задание 19 29

Задание 20 66

Задание 21

```
VAR a1,a2 : Integer;
BEGIN
   ReadLn (a1,a2);
   if a1>a2  then WriteLn (a1+a2)
             else WriteLn (a1*a2);
   WriteLn('ЗАДАЧА РЕШЕНА');
   ReadLn
END.
```

Задание 22

```
VAR a,b,c : Integer;
BEGIN
   ReadLn (a,b,c);
   if a<b+c  then WriteLn ('Подходит.')
             else WriteLn ('Не подходит, слишком длинен.');
   ReadLn
END.
```

Задание 23

```
VAR golov, glaz, N : Integer;
BEGIN
   WriteLn ('Сколько лет дракону?');
   ReadLn (N);
   if N<100 then golov := 3*N
            else golov := 300 + 2*(N-100);
   glaz := 2*golov;
   WriteLn ('У дракона ', golov  ,' голов и ', glaz  ,' глаз');
   ReadLn
END.
```

Задание 24

```
VAR  imya      :String;
     Vozrast   :Integer;
BEGIN
   WriteLn ('Здравствуй, я компьютер, а тебя как зовут?');
   ReadLn (imya);
   WriteLn ('Очень приятно, ', imya, '. Сколько тебе лет?');
   ReadLn (vozrast);
   WriteLn ('Ого! Целых ' ,vozrast, ' лет! Ты уже совсем взрослый!');
   if vozrast<=17  then  begin
      WriteLn ('В какой школе ты учишься?');
      ReadLn;      {Во время паузы вы можете вводить любой текст,
                    все равно он программе не понадобится}
      WriteLn ('Неплохая школа!')
                          end
                  else  begin
      WriteLn ('В каком институте ты учишься?');
      ReadLn;
      WriteLn ('Хороший институт!')
                          end;
   WriteLn ('До следующей встречи!');
   ReadLn
END.
```

Задание 25

```
VAR a,b,c : Integer;
BEGIN
   ReadLn (a,b,c);
   if a>=b+c     then WriteLn ('Неправда')
                 else if b>=a+c  then WriteLn ('Неправда')
                                 else if c>=a+b  then WriteLn ('Неправда')
                                                 else WriteLn ('Правда');
   ReadLn
END.
```

Задание 26 Ей нравятся любые черноглазые, но только не те, у кого рост находится в диапазоне от 180 до 184**.**

Задание 27

```
VAR    a,b    :String;  {a-ПРИВЕТСТВИЕ ЧЕЛОВЕКА, b-ОТВЕТ КОМПЬЮТЕРА}
BEGIN
   ReadLn (a);
   if  a='Здравия желаю'                                        then b:='Вольно';
   if  a='Здорово'                                              then b:='Здравствуйте';
   if (a='Добрый день')OR(a='Приветик')OR(a='Салют')then b:='Салют';
   if (a='Привет')OR(a='Здравствуйте')                          then b:=a;
   WriteLn (b,'!');
   ReadLn
END.
```

Задание 28

```
VAR    bukva : Char;
BEGIN
   WriteLn ('Введи строчную букву русского алфавита');
   ReadLn (bukva);
   case bukva of
```

```
        'а','е','и','о','у','ы','э','ю','я'          :WriteLn('гласная');
        'б','з','в','г','д','ж'                       :WriteLn('согласная звонкая');
        'п','с','ф','к','т','ш'                       :WriteLn('согласная глухая');
        'й','л','м','н','р','х','ц','ч','щ','ъ','ь'  :WriteLn('другая');
        else                                           WriteLn('Таких букв не знаю')
    end;
    ReadLn
END.
```

Задание 29

```
VAR  a,b,rez      : Real;   {a и b - два числа, rez-результат}
     Oper         : Char;   {oper  - знак арифметического действия}
BEGIN
    ReadLn (a);
    ReadLn (oper);
    ReadLn (b);
    case oper of
        '+'      : rez:=a+b;
        '-'      : rez:=a-b;
        '*'      : rez:=a*b;
        '/'      : rez:=a/b;
        else     WriteLn('Таких действий не знаю')
    end;
    WriteLn(rez :11:8);
    ReadLn
END.
```

Задание 30 Эта программа будет печатать:

```
Считаем зайцев
10 зайцев
10 зайцев
11 зайцев
13 зайцев
16 зайцев
20 зайцев
……
```

Операторы *n:=n+1* и *WriteLn('Посчитали зайцев')* не будут выполнены никогда.

Задание 31

```
LABEL m1;   BEGIN   m1: Write ('A');   ReadLn;  goto m1   END.
```

Задание 32

```
LABEL   m1;
VAR     i   :LongInt;
BEGIN
        i:=1000;
 m1:    Write (i,' ');
        ReadLn;
        i:=i-1;
        goto m1
END.
```

Задание 33

```
LABEL   m1;
VAR     a   :Real;
BEGIN
```

```
        a:=100;
 m1:    Write (a :12:8,' ');
        ReadLn;
        a:=a/2;
        goto m1
END.
```

Задание 34

```
LABEL   m1,m2;
VAR     i    :LongInt;
BEGIN
        i:=1;
 m1:    Write (i,' ');
        i:=i+1;
        if i<100 then goto m1;

 m2:    Write (i,' ');
        i:=i-1;
        if i>=1 then goto m2;
        ReadLn
END.
```

Задание 35

```
LABEL   m;
VAR     a   :Real;
BEGIN
        a:=0;
  m:    WriteLn (a :5:3,' ', a*a :9:6);
        a:=a+0.001;
        if a<=1.00001 then goto m;
        ReadLn
END.
```

Пояснение: Вместо *if a<=1 then* я написал *if a<=1.00001 then* и вот по какой причине. Вещественные числа компьютер складывает с незначительной погрешностью, но ее достаточно, чтобы при тысячекратном прибавлении 0.001 набралась не 1, а чуть-чуть больше. А это значит, что счет остановился бы на 0.999. Если не верите, попробуйте распечатывать a с 15 знаками после точки. Подробнее о причинах – см. 12.2

Задание 36

```
LABEL   m1,m2;
VAR     x,y,z   :Real;
BEGIN
        x:=2700;
  m1:   y:=x/4 + 20;
        z:=2*y+0.23;
        WriteLn ('x=',x:12:6,' y=',y:12:6,' z=',z:12:6);
        if y*z<1/x then goto m2;
        x:=x/3;
        goto m1;
  m2:   ReadLn
END.
```

Задание 37

```
VAR  Slovo     :String;
     Nomer     :Integer;
BEGIN
   Nomer:=1;
```

```
    repeat
        WriteLn('Введите слово');
        ReadLn(Slovo);
        WriteLn(Nomer, ' ', Slovo, '!');
        Nomer:=Nomer+1;
    until Slovo='Хватит';
    WriteLn('Хватит так хватит');
    ReadLn
END.
```

Задание 38

```
VAR   a :Real;
BEGIN
    a:=0;
    repeat
        WriteLn (a :5:3,' ', a*a :9:6);
        a:=a+0.001;
    until a>1.00001;
    ReadLn
END.
```

Задание 39

```
VAR   x,y,z :Real;
BEGIN
    x:=8100;
    repeat
        x:=x/3;
        y:=x/4 + 20;
        z:=2*y+0.23;
        WriteLn ('x=',x:12:6,' y=',y:12:6,' z=',z:12:6);
    until y*z<1/x;
    ReadLn
END.
```

Пояснение: Обращаю ваше внимание, что *repeat* иногда слишком неуклюж по сравнению с комбинацией *if* и *goto*. Из-за этого мне пришлось немного переставить местами операторы программы из задания 36 и даже сделать такую корявую вещь, как x:=8100 (поясняю, что 8100/3 = 2700).

Задание 40

```
VAR t,s,h,v : Real;
BEGIN
    v:=20;
    t:=0;
    repeat
        s:= v*t;
        h:= 100-9.81*t*t/2;
        WriteLn('t=',t:5:1,'  s=',s:8:2,'  h=',h:6:2);
        t:=t+0.2;
    until  h<=0;      {Отрицательная высота - значит упал на землю}
    ReadLn
END.
```

Задание 41

```
VAR a : Real;
BEGIN
    a:=900;
    while a>=0 do  begin   {Из отрицательных чисел корни компьютер не вычисляет}
```

```
                    WriteLn('Число=',  a :5:0,   '  Корень=',   Sqrt(a) :7:3);
                    a:=a-3;
                end;
  ReadLn
END.
```

Задание 42

```
VAR i : Integer;
BEGIN
  Write('Прямой счет: ');
  for i:=    -5  to          5 do Write(i,' ');
  Write('Обратный счет: ');
  for i:=     5  downto   -5 do Write(i,' ');
  Write('Конец счета');
  ReadLn
END.
```

Задание 43

```
VAR  i, N, a  : Integer;
BEGIN
  WriteLn('Введите число кубиков');
  ReadLn (N);
  for i:=1 to N do begin
    WriteLn('Введите длину стороны кубика');
    ReadLn (a);
    WriteLn('Объем кубика=', a*a*a)
  end;
  ReadLn
END.
```

Задание 44 Компьютер напечатает:

```
Площадь пола=300 Объем зала=1200
Площадь пола=300 Объем зала=1200
Площадь пола=300 Объем зала=1200
```

и не спросит размеры 2 и 3 залов**.**

Задание 45 Компьютер напечатает результаты только для последнего зала**.**

Задание 46 Компьютер напечатает результат:

на 10 больше правильного*.*

в два раза больше правильного**.**

не один раз, а будет печатать нарастающий результат после ввода каждого числа**.**

0 или 1, так как на каждом цикле счетчик будет обнуляться**.**

200 или 0 в зависимости от того, положительно первое число или нет**.**

Задание 47

```
VAR     i, a, N, c_pol, c_otr, c_10 : Integer;
BEGIN
  WriteLn('Введите количество чисел');
  ReadLn (N);
  c_pol:=0; c_otr:=0; c_10 :=0;          {Обнуляем счетчики}
  for i:=1 to N do begin
    WriteLn('Введите число');
    ReadLn (a);
    if a>0      then c_pol:=c_pol+1;      {Подсчитываем положительные}
    if a<0      then c_otr:=c_otr+1;      {Подсчитываем отрицательные}
```

```
        if a>10    then c_10 :=c_10 +1;      {Подсчитываем превышающие 10}
    end {for};
    WriteLn('Положит - ',c_pol,'  Отрицат - ',c_otr,'  Больших 10 - ',c_10);
    ReadLn
END.
```

Задание 48

```
VAR     a, b, c : Integer;
BEGIN
    c:=0;                          {Обнуляем счетчик}
    repeat
        ReadLn (a,b);              {Ввод пары чисел}
        if a+b=13 then c:=c+1;
    until (a=0) AND (b=0);         {пока не введена пара нулей}
    WriteLn(c);
    ReadLn
END.
```

Задание 49

18

10

5 и 8

3

10

3

5

Задание 50

```
VAR i, dlina, shirina, S, sum: Integer;
BEGIN   sum:=0;
    for i:=1 to 40 do begin
        ReadLn (dlina, shirina);
        S:=dlina*shirina;          {S-площадь зала}
        sum:=sum+S                 {sum-площадь дворца}
    end {for};
    WriteLn(sum);
    ReadLn
END.
```

Задание 51

```
VAR  i, ball, N, S : Integer;
BEGIN
    WriteLn('Введите количество учеников');
    ReadLn (N);
    S:=0;
    for i:=1 to N do begin
        WriteLn('Введите балл ученика');
        ReadLn (ball);
        S:=S+ball;
    end;
    WriteLn('Средний балл =',S/N :8:3);
    ReadLn
END.
```

Задание 52

```
VAR  i, N                : Integer;
     a, proizvedenie : Real;
BEGIN
   WriteLn('Введите количество сомножителей');
   ReadLn (N);
   proizvedenie:=1;                          {Сумму обнуляем, произведение - нет!}
   for i:=1 to N do begin
      WriteLn('Введите сомножитель');
      ReadLn (a);
      proizvedenie := proizvedenie * a;    {Наращиваем произведение}
   end;
   WriteLn('Произведение =',proizvedenie :12:3);
   ReadLn
END.
```

Задание 53

```
VAR  perv, vtor : Integer;       {первая и вторая цифры}
BEGIN
   for perv:=3 to 8 do  for vtor:=0 to 7 do Write(perv,vtor,' ');
   ReadLn
END.
```

Задание 54

```
VAR  i,j,k,l : Integer;       {четыре цифры}
BEGIN
   for i:=1 to 3 do
      for j:=1 to 3 do
         for k:=1 to 3 do
            for l:=1 to 3 do
               Write(i,j,k,l,' ');
   ReadLn
END.
```

Задание 55

```
VAR  i,j,k,l,  c   : Integer;  {c-счетчик}
BEGIN
   c:=0;                          {Обнуляем счетчик}
   for i:=1 to 3 do for j:=1 to 3 do for k:=1 to 3 do for l:=1 to 3 do c:=c+1;
   Write('Количество сочетаний = ', c);
   ReadLn
END.
```

Задание 56

```
VAR  i,j,k,l,  c   : Integer;  {c-счетчик}
BEGIN
   c:=0;                          {Обнуляем счетчик}
   for i:=1 to 3 do
      for j:=1 to 3 do
         for k:=1 to 3 do
            for l:=1 to 3 do
               if (i<=j) AND (j<=k) AND (k<=l) then c:=c+1;
   WriteLn('Количество неубывающих сочетаний = ', c);
   ReadLn
END.
```

Задание 57

```
VAR   i,N,  chislo,  min, nomer   :Integer;
BEGIN
   WriteLn('Введите количество чисел');
   ReadLn (N);                        {N - количество чисел}
   ReadLn(min);                       {первое число считаем минимальным}
   nomer:=1;                          {его номер - первый}
   for i:=2 to N do begin             {Просматриваем остальные числа}
      ReadLn(chislo);
      if  chislo<min  then begin      {Если число меньше минимального, то}
         min:=chislo;                 {оно становится минимальным}
         nomer:=i;                    {запоминаем номер минимального числа}
      end {if};
   end {for};
   WriteLn(min,' ',nomer);
   ReadLn
END.
```

Задание 58

```
VAR   i,N,  rost,  min, max   :Integer;
BEGIN
   WriteLn('Сколько человек в классе?');
   ReadLn (N);
   max:=0;                    {Ясно, что роста меньше 0 см не бывает}
   min:=500;                  {Ясно, что роста больше 500 см не бывает}
   for i:=1 to N do begin     {Просматриваем все числа}
      WriteLn('Введите рост ученика');
      ReadLn(rost);
      if  rost<min      then min:=rost;
      if  rost>max      then max:=rost
   end{for};
   if max-min>40 then    WriteLn('Правда') else WriteLn('Неправда');
   ReadLn
END.
```

Задание 60

```
USES CRT;
VAR hz, i  : Integer;
BEGIN
   for i:=1 to 3 do begin             {Повторить три раза звук сирены}
      hz:=60;
      while hz<800 do begin           {Звук вверх}
         Sound(hz); Delay(50);
         hz:=hz+5
      end;
      while hz>60 do begin            {Звук вниз}
         Sound(hz); Delay(50);
         hz:=hz-5
      end;
   end{for};
   NoSound
END.
```

Задание 61

```
USES CRT;
VAR hz, i  : Integer;
BEGIN
   for i:=1 to 30 do begin
         Sound(60);
```

```
            Delay(50);
            Sound(400);
            Delay(50);
    end{for};
    NoSound
END.
```

Задание 62

```
USES CRT;
VAR hz: Integer;
BEGIN
    hz:=1000;
    while hz<20000 do begin
        WriteLn('Частота звука - ', hz, ' герц. Жмите кл.ввода до 20000 гц.');
        Sound(hz);
        ReadLn;
        hz:=hz+500
    end;
    NoSound
END.
```

Задание 64

```
USES CRT;
PROCEDURE doo;    BEGIN Sound(523); Delay(500); NoSound; Delay(20) END;
PROCEDURE re;     BEGIN Sound(587); Delay(500); NoSound; Delay(20) END;
PROCEDURE mi;     BEGIN Sound(659); Delay(500); NoSound; Delay(20) END;
PROCEDURE fa;     BEGIN Sound(698); Delay(500); NoSound; Delay(20) END;
PROCEDURE sol;    BEGIN Sound(784); Delay(500); NoSound; Delay(20) END;
PROCEDURE la;     BEGIN Sound(880); Delay(500); NoSound; Delay(20) END;
PROCEDURE si;     BEGIN Sound(988); Delay(500); NoSound; Delay(20) END;
      {500 - продолжительность звука, 20 - пауза между нотами}
BEGIN
    mi; doo; mi; doo; fa; mi; re; sol; sol; la; si; doo; doo; doo
END.
```

Задание 65

```
USES CRT;
PROCEDURE doo;    BEGIN Sound(523); Delay(500); NoSound; Delay(20) END;
PROCEDURE re;     BEGIN Sound(587); Delay(500); NoSound; Delay(20) END;
PROCEDURE mi;     BEGIN Sound(659); Delay(500); NoSound; Delay(20) END;
PROCEDURE fa;     BEGIN Sound(698); Delay(500); NoSound; Delay(20) END;
PROCEDURE sol;    BEGIN Sound(784); Delay(500); NoSound; Delay(20) END;
PROCEDURE la;     BEGIN Sound(880); Delay(500); NoSound; Delay(20) END;
PROCEDURE si;     BEGIN Sound(988); Delay(500); NoSound; Delay(20) END;
PROCEDURE chijik;
    BEGIN mi; doo; mi; doo; fa; mi; re; sol; sol; la; si; doo; doo; doo END;
BEGIN
    WriteLn('Песня "Чижик-пыжик". 1 куплет');
    chijik;
    WriteLn('2 куплет');
    chijik;
END.
```

Задание 66

```
Я, король Франции, спрашиваю вас - кто вы такие? Вот ты - кто такой?
Я - Атос
А ты, толстяк, кто такой?
```

```
А я Портос! Я правильно говорю, Арамис?
Это так же верно,как то,что я -Арамис!
Он не врет, ваше величество! Я Портос, а он Арамис.
А ты что отмалчиваешься, усатый?
А я все думаю, ваше величество - куда девались подвески королевы?
Анна! Иди-ка сюда!!!
```

Задание 67

```
USES Graph;
VAR  Device, Mode: Integer;
BEGIN
   Device:=0;
   InitGraph(Device, Mode, 'c:\tp\bgi');
   Rectangle(300,30,360,80);      {шапка}
   Circle(330,120,40);            {голова}
   Circle(345,110,5);             {глаз}
   Circle(315,110,5);             {глаз}
   Line(320,140,340,140);         {рот}
   Line(330,120,330,130);         {нос}
   Line(330,120,305,130);         {нос}
   Line(330,130,305,130);         {нос}
   Circle(330,220,60);            {середина}
   Circle(330,360,80);            {низ}
   Rectangle(350,163,455,183);    {рука}
   Rectangle(203,163,308,183);    {рука}
   Line(210,130,210,440);         {посох}
   ReadLn;
   CloseGraph
END.
```

Задание 68

```
USES Graph;
VAR  Device, Mode: Integer;
BEGIN
   Device:=0;
   InitGraph(Device, Mode, 'c:\tp\bgi');
   Rectangle(300,30,360,80);      {шапка}
   SetFillStyle(1, yellow);       {заливка}
   FloodFill(330,50, white);      {шапки}
   Circle(330,120,40);            {голова}
   Circle(345,110,5);             {глаз}
   Circle(315,110,5);             {глаз}
   SetColor(red);
   Line(320,140,340,140);         {рот}
   SetColor(white);
   Line(330,120,330,130);         {нос}
   Line(330,120,305,130);         {нос}
   Line(330,130,305,130);         {нос}
   SetFillStyle(1, red);          {заливка}
   FloodFill(328,125, white);     {носа}
   Circle(330,220,60);            {середина}
   Circle(330,360,80);            {низ}
   Rectangle(350,163,455,183);    {рука}
   Rectangle(203,163,308,183);    {рука}
   SetLineStyle(0, 0, ThickWidth);
   SetColor(blue);
   Line(210,130,210,440);         {посох}
   WriteLn('Это снеговик');
   ReadLn;
   CloseGraph
END.
```

Задание 69

```
x:=x+4;
```

Задание 70

```
x:=40;
repeat
    Circle(x,100,10);
    x:=x+4;
until x>600;
```

Задание 71

```
    Circle(x,100,40);
```

Задание 72

```
USES Graph;
VAR x,y, Device, Mode :Integer;
BEGIN
   Device:=0;
   InitGraph(Device,   Mode,    'c:\tp\bgi');
   x:=40;
   y:=470;
   repeat
      PutPixel(x,y,white);
      x:=x+20;
      y:=y-15
   until x>600;
   ReadLn;
   CloseGraph
END.
```

Задание 73

```
USES Graph;
VAR r, Device, Mode :Integer;
BEGIN
   Device:=0;
   InitGraph(Device,   Mode,    'c:\tp\bgi');
   r:=10;
   repeat
      Circle(320,240,r);
      r:=r+15;
   until r>230;
   ReadLn;
   CloseGraph
END.
```

Задание 74

```
   SetColor(Yellow);
   r:=50;
   repeat
      Circle(320,240,r);
      r:=r+2;
   until r>230;
```

Задание 75

```
y:=120;
r:=0;
repeat
    Circle(320,y,r);
    r:=r+3;
    y:=y+2;
until r>200;
```

Задание 76

```
x:=40;
y:=40;
r:=0;
repeat
    Circle(x,y,r);
    x:=x+4;
    y:=y+2;
    r:=r+1;
until x>500;
```

Задание 77

```
y:=10;
repeat
    Line(0,y,640,y);
    y:=y+10;
until y>480;
```

Задание 78

```
y:=10;
repeat          {горизонтальные линии:}
    Line(0,y,640,y);
    y:=y+10;
until y>480;
x:=10;
repeat          {вертикальные линии:}
    Line(x,0,x,480);
    x:=x+10;
until x>640;
```

Задание 79

```
y:=10;
repeat                  {горизонтальные линии:}
    Line(0,y,640,y);
    y:=y+10;
until y>480;
x:=10;
repeat                  {наклонные линии:}
    Line(x,0,x-100,480); {x-100 означает, что нижний конец любой линии}
                         {будет на 100 пикселов левее верхнего}
    x:=x+10;
until x>800;            {мы можем рисовать и за пределами экрана}
```

Задание 80

```
x:=50;
```

```
    repeat
        Rectangle(x,100,x+40,140);
            {Верхняя и нижняя стороны квадрата остаются всегда на одной высоте
            (100 и 140). Горизонтальные координаты левого верхнего (x) и правого
            нижнего (x+40) углов меняются:}
        x:=x+50;
    until x>580;
```

Задание 81

```
USES Graph;
VAR  i,j, x,y, Device,Mode :Integer;
BEGIN
    Device:=0;
    InitGraph(Device, Mode, 'c:\tp\bgi');
    y:=80;                              {горизонтальные линии:}
    repeat    Line(160,y,480,y);
              y:=y+40;
    until y>400;
    x:=160;                             {вертикальные линии:}
    repeat    Line(x,80,x,400);
              x:=x+40;
    until x>480;
    Rectangle(155,75,485,405);          {Рамка вокруг доски}
                                        {Закрашиваем клетки в шахматном порядке:}
    SetFillStyle(1,Yellow);
    y:=100;                             {центр верхнего ряда}
    for i:=1 to 4 do begin              {четыре пары рядов клеток}
        x:=180;                         {центр самого левого столбца}
        for j:=1 to 4 do begin          {закрашиваем нечетный ряд клеток}
            FloodFill(x,y,White);
            x:=x+80                     {перескакиваем через клетку направо}
        end{for};
        y:=y+40;                        {перескакиваем вниз, в четный ряд клеток}
        x:=220;                         {центр второго слева столбца}
        for j:=1 to 4 do begin          {закрашиваем четный ряд клеток}
            FloodFill(x,y,White);
            x:=x+80                     {перескакиваем через клетку направо}
        end{for};
        y:=y+40;                        {перескакиваем вниз, в нечетный ряд клеток}
    end{for};
    ReadLn;
    CloseGraph
END.
```

Задание 82

```
USES Graph;
VAR  x,y, Device,Mode :Integer;
BEGIN
    Device:=0;
    InitGraph(Device, Mode, 'c:\tp\bgi');
    y:=40;
    repeat
        x:=40;
        repeat          {рисуем горизонтальный ряд окружностей:}
            Circle(x,y,20);
            x:=x+12;
        until x>600;
        y:=y+12;        {перескакиваем вниз к следующему ряду:}
    until y>440;
    ReadLn;
    CloseGraph
```

```
END.
```

Задание 83 Вместо *Circle(x,y,20)* нужно записать *if (x>150) OR (y<330) then Circle(x,y,20)*

Задание 84 Вместо *Circle(x,y,20)* нужно записать

```
if      ((x>150) OR (y<330))
                AND
        ((x<260) OR (x>380) OR (y<180) OR (y>300))
then Circle(x,y,20)
```

Задание 85

```
USES Graph;
VAR  i, Device,Mode :Integer;
BEGIN
   Device:=0;
   InitGraph(Device, Mode, 'c:\tp\bgi');
   for i:=1 to 30 do Circle(Random(640),Random(480),20);
   ReadLn;
   CloseGraph
END.
```

Задание 86

```
   for i:=1 to 100 do begin
      Circle(Random(640),Random(480),Random(100));
      SetColor(Random(15))
   end{for};
```

Задание 87

```
USES Graph;
VAR  i, Device,Mode :Integer;
BEGIN
   Device:=0;
   InitGraph(Device, Mode, 'c:\tp\bgi');
   Rectangle(300,100,400,250);                    {окно}
   for i:=1 to 100 do PutPixel(300+Random(100), 100+Random(150), Random(16));
   ReadLn;
   CloseGraph
END.
```

Задание 89

```
USES Graph, CRT;
VAR x, Device, Mode: Integer;
BEGIN
   Device:=0;
   InitGraph(Device, Mode, 'c:\tp\bgi');
   ReadLn; {Пауза на секундочку, чтобы успел установиться графический режим}
   x:=40;
   repeat
      SetColor(White);
      Circle(x,100,10);    {Рисуем окружность}
      Circle(x,200,10);    {Рисуем вторую окружность}
      Delay(10);
      SetColor(Black);
      Circle(x,100,10);    {Стираем окружность}
      Circle(x,200,10);    {Стираем вторую окружность}
```

```
    x:=x+1              {Перемещаемся немного направо}
  until x>600;          {пока не упремся в край экрана}
  CloseGraph
END.
```

Задание 90

```
  x:=40; y:=40;
  repeat
    SetColor(White);
    Circle(x,100,10);   {Рисуем окружность}
    Circle(100,y,10);   {Рисуем вторую окружность}
    Delay(10);
    SetColor(Black);
    Circle(x,100,10);   {Стираем окружность}
    Circle(100,y,10);   {Стираем вторую окружность}
    x:=x+1; y:=y+1;     {Перемещаемся}
  until x>600;          {Пока не упремся в край экрана}
```

Задание 91

```
  x:=40;
  repeat                {Движемся направо}
    SetColor(White);  Circle(x,100,10);
    Delay(10);
    SetColor(Black);  Circle(x,100,10);
    x:=x+1;
  until x>600;          {Пока не упремся в правый край экрана}
  repeat                {Движемся налево}
    SetColor(White);  Circle(x,100,10);
    Delay(10);
    SetColor(Black);  Circle(x,100,10);
    x:=x-1;
  until x<40;           {Пока не упремся в левый край экрана}
```

Задание 92

"Обнимите" весь вышеприведенный фрагмент из задания 91 конструкцией *repeat ..... until 2>3;*

Задание 93

```
USES Graph, CRT;
VAR x,y, dx,dy,  Device, Mode: Integer;   {dx - шаг шарика по горизонтали,
  то есть расстояние по горизонтали между двумя последовательными
  изображениями окружности. dy - аналогично по вертикали}
BEGIN
  Device:=0;
  InitGraph(Device, Mode, 'c:\tp\bgi');
  Rectangle(35,35,605,445);       {бортики стола}
  x:=320;  y:=240;                {Начинаем движение шарика из центра}
  dx:=1; dy:=1;                   {Направление движения - вправо вниз}
  repeat
    SetColor(White);  Circle(x,y,10);
    Delay(10);
    SetColor(Black);  Circle(x,y,10);
    x:=x+dx; y:=y+dy;

    if (x<50)OR(x>590)    then dx:=-dx;    {Ударившись о левый или правый борт,
      шарик меняет горизонтальную составляющую скорости на противоположную}
    if (y<50)OR(y>430)    then dy:=-dy;    {Ударившись о верхний или нижний борт,
      шарик меняет вертикальную составляющую скорости на противоположную}

    if          (x<80)    AND  (y<80)      {Если шарик в левом верхнем углу}
```

```
          OR   (x<80)   AND  (y>400)      {или в левом нижнем}
          OR   (x>560)  AND  (y<80)       {или в правом верхнем}
          OR   (x>560)  AND  (y>400)      {или в правом нижнем,}
        then                              {то прорисовывай шарик и делай паузу:}
          begin SetColor(White); Circle(x,y,10); ReadLn; Halt end;

   until 2>3;
END.
```

Задание 94

```
USES Graph, CRT;
VAR  x,y, x0,y0, Device,Mode  : Integer;
     t,s,h,v                  : Real;
BEGIN
   Device:=0;
   InitGraph(Device, Mode, 'c:\tp\bgi');
   Rectangle(20,40,40,440);                    {башня}
   Line(0,440,640,440);                        {земля}
   x0:=40;   y0:=40;                           {Координаты верха башни}
   v:=20;     t:=0;                            {Начальные скорость и время}
   ReadLn;                                     {Пауза перед броском}
   repeat
      s:= 4*v*t;          h:= 4*(100-9.81*t*t/2);
      x:=x0+Round(s);     y:= 400+y0-Round(h);{Округляю, так как процедура
                                               Circle(x,y,3) требует целых x и y}
      t:=t+0.05;
      SetColor(White);  Circle(x,y,3);
      PutPixel(x,y,white);                     {след от камня}
      Delay(100);
      SetColor(Black);  Circle(x,y,3);
   until h<0;
   SetColor(White);  Circle(x,y,3);            {Прорисовываем камень последний раз}
   ReadLn;
   CloseGraph
END.
```

Задание 96

```
USES Graph, CRT;
VAR  Device, Mode, x,r, y_red, y_yellow, y_green  : Integer;
     klavisha                                     : Char;
BEGIN
   Device:=0;
   InitGraph(Device, Mode, 'c:\tp\bgi');

   x          :=320;       {задаем центр светофора по горизонтали}
   r          := 50;       {задаем радиус огней светофора}
   y_red      :=110;       {задаем центр красного огня по вертикали}
   y_yellow   :=240;       {задаем центр желтого огня по вертикали}
   y_green    :=370;       {задаем центр зеленого огня по вертикали}

   Rectangle(x-100,40,x+100,440);          {рисуем светофор}
   Circle(x,y_red,    r);
   Circle(x,y_yellow,r);
   Circle(x,y_green, r);

   repeat
    if KeyPressed then begin      {Если нажата какая-нибудь клавиша, то:}
      SetFillStyle(1,Black);      {прежде всего гасим:}
      FloodFill(x,y_red,   White);{верхний огонь, даже если он не горел}
      FloodFill(x,y_yellow,White);{средний огонь, даже если он не горел}
      FloodFill(x,y_green, White);{нижний  огонь, даже если он не горел}
```

```
      klavisha:= ReadKey;
      if klavisha='r' then          {если была нажата r, то зажигаем красный:}
         begin SetFillStyle(1,red);      FloodFill(x,y_red,   White)      end;
      if klavisha='y' then          {если была нажата y, то зажигаем желтый:}
         begin SetFillStyle(1,yellow);   FloodFill(x,y_yellow,White)      end;
      if klavisha='g' then          {если была нажата g, то зажигаем зеленый:}
         begin SetFillStyle(1,green);    FloodFill(x,y_green, White)      end;
    end{if}
  until klavisha='q';               {если была нажата q, то выходим из пр-мы}
  CloseGraph
END.
```

Задание 97

```
USES Graph,CRT;
VAR x,y, Device, Mode: Integer;
BEGIN
  Device:=0;
  InitGraph(Device,  Mode,   'c:\tp\bgi');
  ReadLn;

  x:=750;                           {Задаем начальную координату самолета}
  repeat                            {Самолет летит в одиночку...}
    SetColor(White);
    Ellipse(x,100,0,360,50,10);
    Delay(20);
    SetColor(Black);
    Ellipse(x,100,0,360,50,10);
    x:=x-1
  until KeyPressed;                 {до тех пор, пока не будет нажата любая клавиша,
                                     после чего самолет и снаряд летят одновременно:}
  y:=500;                           {Задаем начальную координату снаряда}
  repeat
    SetColor(White);
    Ellipse(x,100,0,360,50,10);     {рисуем самолет}
    Ellipse(50,y,0,360,5,10);       {рисуем снаряд}
    Delay(20);
    SetColor(Black);
    Ellipse(x,100,0,360,50,10);     {стираем самолет}
    Ellipse(50,y,0,360,5,10);       {стираем снаряд}
    x:=x-1;                         {перемещаем самолет}
    y:=y-1                          {перемещаем снаряд}
  until y<0;                        {до тех пор, пока снаряд не долетит до верха экрана}
  CloseGraph
END.
```

Задание 98-99

```
USES Graph, CRT;
VAR  Device, Mode, x, y, d  : Integer;
     klavisha               : Char;
BEGIN
  Device:=0;
  InitGraph(Device, Mode, 'c:\tp\bgi');

  x  :=320;                         {Задаем начальные координаты точки}
  y  :=240;
  d  :=5;                             {Задаем шаг перемещения точки}
  PutPixel(x,y,White);                {Рисуем точку в начальном положении}
  repeat
   if KeyPressed then begin           {Если нажата какая-нибудь клавиша, то:}
    PutPixel(x,y,Black);              {стираем точку в старом положении}
    klavisha:= ReadKey;
```

```
      if klavisha='d'   then x:=x+d;     {если нажата d, то шаг направо}
      if klavisha='a'   then x:=x-d;     {если нажата a, то шаг налево}
      if klavisha='z'   then y:=y+d;     {если нажата z, то шаг вниз}
      if klavisha='w'   then y:=y-d;     {если нажата w, то шаг вверх}
      if klavisha='m'   then d:=d+1;     {если нажата m, то шаг увеличиваем}
      if (klavisha='l') AND (d>0)        {если нажата l и шаг еще положителен,}
                        then d:=d-1;     {то шаг уменьшаем}
      PutPixel(x,y,White);               {рисуем точку в новом положении}
    end{if}
  until klavisha='q';                    {если была нажата q, то выходим из пр-мы}
  CloseGraph
END.
```

Интересная возможность: Уберите одну из PutPixel - и точка начнет оставлять за собой след, то есть "рисовать" - вы получите простейший "графический редактор"**.**

Задание 102 **1)** a[i] = a[i-1] + 4 **2)** a[i] = 2 * a[i-1] **3)** a[i] = 2 * a[i-1] - 1

Задание 103

```
{Эта программа практически копирует программу про длину тысячи удавов,
так как среднее значение равняется сумме, деленной на число слагаемых:}
VAR  t   :array [1..7] of Integer;   {t - массив температур за 7 дней}
     s,i :Integer;                   {s - сумма}
BEGIN  {Зададем температуры присвоением:}
  t[1]:=-21; t[2]:=-12; t[3]:=0; t[4]:=4; t[5]:=-5; t[6]:=-14; t[7]:=-24;
     {Суммируем весь массив значений температур:}
  s:= 0;
  for i:=1 to 7 do s:=s+t[i];
  WriteLn('Средняя температура = ', s/7 : 6:2);
  ReadLn
END.
```

Задание 104

```
VAR  t   :array [1..7] of Integer;   {t - массив температур за 7 дней}
     c,i :Integer;                   {c - счетчик морозных дней}
BEGIN   {Зададем температуры присвоением:}
  t[1]:=-21; t[2]:=-12; t[3]:=0; t[4]:=4; t[5]:=-5; t[6]:=-14; t[7]:=-24;
  c:= 0;
  for i:=1 to 7 do if t[i]<-20 then c:=c+1;
  WriteLn('Морозных дней было ', c);
  ReadLn
END.
```

Задание 105

```
  min:=t[1];
  for i:=2 to 7 do if t[i]<min then  begin min:=t[i]; nomer:=i end;
  WriteLn('Номер самого морозного дня - ', nomer);
```

Задание 106

```
VAR  f   :array [1..30] of LongInt;
     I   :Integer;
BEGIN
  f[1]:=1; f[2]:=1;
  for i:=3 to 30 do begin f[i] := f[i-1] + f[i-2]; Write(' ', f[i]) end;
  ReadLn
END.
```

Задание 107

```
VAR  t              :array [1..3, 1..4] of Integer;
     i,j,min,max    :Integer;
BEGIN
   t[1,1]:=-8;    t[1,2]:=-14;   t[1,3]:=-19;   t[1,4]:=-18;
   t[2,1]:=25;    t[2,2]:= 28;   t[2,3]:= 26;   t[2,4]:= 20;
   t[3,1]:=11;    t[3,2]:= 18;   t[3,3]:= 20;   t[3,4]:= 25;
       {За первое значение максимума и минимума примем первое из проверяемых чисел:}
   min:= t[1,1];
   max:= t[1,1];
   for i:=1 to 3 do
      for j:=1 to 4 do begin
         if t[i,j]<min   then min:=t[i,j];
         if t[i,j]>max   then max:=t[i,j]
      end{for};
   WriteLn (max-min);
   ReadLn
END.
```

Задание 108

```
              {Вариант 1}
VAR  t1_den,   t2_den,      t_den     :1..30;   {t1 - время отправления, t2 - время}
     t1_chas,  t2_chas,     t_chas    :0..23;   {прибытия, t - время в пути, den - }
     t1_min,   t2_min,      t_min     :0..59;   {день, chas - часы, min - минуты}
     minut,    minut1                :Word;
BEGIN
   WriteLn('Введите время отправления(день месяца, час, минута через пробел)');
   ReadLn(t1_den, t1_chas, t1_min);
   WriteLn('Введите время в пути (дни, часы и минуты через пробел)');
   ReadLn(t_den, t_chas, t_min);
         {Сколько минут прошло с 0 часов дня отправления до момента прибытия:}
   minut := 24*60*t_den + 60*(t1_chas+t_chas) + (t1_min+t_min);          {В сутках - 24*60 минут}
         {Вычисляем дату прибытия:}
   t2_den := t1_den + minut DIV (24*60);
         {Сколько минут прошло с 0 часов дня прибытия до момента прибытия:}
   minut1 := minut MOD (24*60);
         {Вычисляем час прибытия:}
   t2_chas := minut1 DIV 60;
         {Вычисляем минуту прибытия:}
   t2_min  := minut1 MOD 60;
   WriteLn('Пароход прибывает в Астрахань ', t2_den,' июня в ', t2_chas, ' час. ', t2_min,' мин.');
   ReadLn
END.
```

Задание 109

```
      BEGIN  WriteLn (Ord('Ф') - Ord('Б') + 1)  END.
```

Задание 110

```
TYPE     mes =(january, february, march, april, may, june, july, august,
               september, october, november, december);
BEGIN
   if september > june  then WriteLn('Правда') else WriteLn('Неправда');
   ReadLn
END.
```

Задание 111

```
TYPE    Ochered =   (Nina, Olga, Alex, Marianna, Ester, Misha, Tolik, Lena,
                     Oleg, Anton, Pankrat, Robocop, Dima, Donatello, Zina,
                     Sveta, Artur,  Ramona, Vera, Igor, Ira);
CONST   money    : array [Nina..Ira] of Word =
                       (5,3,4,7,9,3,6,2,0,3,4,1,1,7,2,7,9,4,5,6,4);
      {Можно было написать не  array [Nina..Ira], а  array [Ochered]}
VAR     i        : Nina..Ira; {Можно было написать не  Nina..Ira, а  Ochered}
        s        : Integer;
BEGIN
   s:=0;                                  {Обнуляем сумматор денег}
   for i:=Nina to Ira do s:=s+money[i];   {суммируем деньги}
   if s>=300    then WriteLn('Хватит')
                else WriteLn('Не хватит');
   WriteLn('Номер Лены в очереди равен ', Ord(Lena)+1);
   if money[Pankrat] > money[Misha]
                then WriteLn('Правда')
                else WriteLn('Неправда');
   ReadLn
END.
```

Задание 112 Компьютер напечатает символ **+**

Задание 113

```
VAR i :Integer;
BEGIN
   for i:=32 to 255 do Write(chr(i),' ');
   ReadLn
END.
```

Задание 114

```
VAR  s   :String;
     i   :Integer;
BEGIN
   s:='Корова';
   for i:=1 to Length(s) div 2 do begin   {Length(s) div 2 - это число пар букв в слове}
      Write(s[2*i-1],s[2*i]);             {Печатаем очередную пару букв}
      Write('быр');
   end{for};
                                          {Допечатываем последнюю нечетную букву, если она есть:}
   if Length(s) mod 2 = 1 then Write(s[Length(s)]);
   ReadLn
END.
```

Задание 115

```
VAR  ishodn, rezult   :String;   {Исходная и результирующая строки}
     i                :Integer;
BEGIN
   ishodn:='Печка';
   rezult:='    '; {Это сделать необходимо, иначе не работает rezult[i]:=}
   for i:=1 to Length(ishodn) do rezult[i]:=chr(Ord(ishodn[i])+1);
   WriteLn(rezult);
   ReadLn
END.
```

Задание 116

```
TYPE    Family =    record
```

```
            imya          :String;
            god_rozd      :Word;
            tsvet_glaz    :String;
                    end;
CONST  me    :Family      =                {me - это я}
            (imya:'Роберт';     god_rozd:1984;     tsvet_glaz:'Серый');
       uncle :Family      =                {дядя}
            (imya:'Сэм';        god_rozd:1940;     tsvet_glaz:'Карий');
       aunt  :Family      =                {тетя}
            (imya:'Салли';      god_rozd:1950;     tsvet_glaz:'Синий');
VAR  i   :   Integer;
BEGIN  {Предположим, на дворе - 1999 год}
   WriteLn (1999 - me.god_rozd,' ',me.tsvet_glaz);
   if  uncle.god_rozd < aunt.god_rozd     then WriteLn('Правда')
                                          else WriteLn('Неправда');
   ReadLn
END.
```

Задание 118

```
CONST   kol  = 10;
VAR     bukvi  :set of 'А'..'Я';
        i      :Integer;
BEGIN
   Randomize;     {Формируем случайным образом множество bukvi}
   bukvi:=[ ];    {Начинаем формировать "с нуля"}
   for i:= 1 to kol do bukvi := bukvi + [chr(Ord('A')+Random(32+1))];
        {Наращиваем по одной букве. Здесь 32 - количество заглавных русских
        букв в таблице ASCII,   Ord('A')+Random(32+1) - случайный номер
        такой буквы в этой таблице}
   if ('М' in bukvi) OR ('И' in bukvi) OR ('Ф' in bukvi)
                                  then WriteLn('Входят')
                                  else WriteLn('Не входят');
   ReadLn
END.
```

Задание 119

```
USES Graph;
VAR x,y,razmer, Device, Mode :Integer;
PROCEDURE treugolnik(x,y,razmer:Integer);
   BEGIN  Line (x,           y,  x+razmer,       y);
          Line (x,           y,  x+razmer div 2, y-razmer);
          Line (x+razmer, y,  x+razmer div 2, y-razmer);
   END;
BEGIN
   Device:=0;
   InitGraph(Device,   Mode,    'c:\tp\bgi');
   treugolnik(320,240,100);
   treugolnik(200,100,20);
   ReadLn;
END.
```

Задание 120

```
FUNCTION Power(Osnovanie:Real;  Stepen:Word) : Real;
 VAR  a:Real;  i:Word;
 BEGIN a:=1;
   for i:=1 to Stepen do a:=a*Osnovanie;        {Здесь нельзя было написать
      Power:=Power*Osnovanie, так как в правой части оператора присвоения
      функция Power обязана быть записана с параметрами}
   Power:=a
```

```
  END;
BEGIN
    WriteLn(Power( 5,2) : 30:10);
    WriteLn(Power(23,0): 30:10);
    ReadLn
END.
```

Задание 121

```
USES Graph;
FUNCTION x(x_nov:Integer):Integer;  BEGIN x := x_nov + 320   END;
FUNCTION y(y_nov:Integer):Integer;  BEGIN y := 240 - y_nov   END;
VAR d,m:Integer;
BEGIN
    d:=0;
    InitGraph(d,m,'c:\tp\bgi');
    Circle(x(310),y(230),10);   {кружок в правом верхнем углу экрана}
    PutPixel(x(0),y(0),White);  {точка в центре экрана}
    ReadLn
END.
```

Задание 122

```
TYPE vector = array [1..5] of Byte;
FUNCTION  max (c:vector) :Byte;
    VAR i,m :Integer;
    BEGIN m:=c[1]; for i:=2 to 5 do  if c[i]>m then m:=c[i]; max:=m END;
FUNCTION  min (c:vector) :Byte;
    VAR i,m :Integer;
    BEGIN m:=c[1]; for i:=2 to 5 do  if c[i]<m then m:=c[i]; min:=m END;
FUNCTION  raznitsa (c:vector) :Byte;
    BEGIN raznitsa := max(c)-min(c) END;
CONST  a   :vector    = (4,2,3,5,5);        {оценки в классе a}
       b   :vector    = (4,3,3,4,3);        {оценки в классе b}
BEGIN
    if raznitsa(a) > raznitsa(b)    then WriteLn('Ровнее учится класс b')
                                    else WriteLn('Ровнее учится класс a');
    ReadLn
END.
```

Задание 123

```
CONST k=7;
TYPE vector = array [1..k] of Integer;
PROCEDURE  termo (var c:vector; popravka:ShortInt);
    VAR i,m :Integer;
    BEGIN for i:=1 to k do  c[i]:=c[i]+popravka END;
CONST  a:vector = (14,12,13,15,15,12,13);  {Показания термометров на станции a}
       b:vector = (-4,-3,-3,-4,-3,-2,0);       {Показания термометров на станции b}
VAR i:Word;
BEGIN
    termo (a,-2);
    WriteLn('Настоящие значения температур на станции a:');
    for i:=1 to k do WriteLn(a[i]);

    termo (b,3);
    WriteLn('Настоящие значения температур на станции b:');
    for i:=1 to k do WriteLn(b[i]);
    ReadLn
END.
```

Задание 124

```
FUNCTION fib(N: Word): LongInt;
  BEGIN
    if N=1 then fib :=1;
    if N=2 then fib :=1;
    if N>2 then fib :=fib(N-2)+fib(N-1)
  END;
VAR i:Word;
BEGIN
    for i:=1 to 35 do Write(fib(i),' ');
    ReadLn
END.
```

Обратите внимание, как долго Паскаль вычисляет последние из чисел Фибоначчи**.** Это - плата за рекурсию**.**

Задание 125

```
{Самый простой способ - преобразовать (вытянуть) двумерный массив в
одномерный, отсортировать его, а затем снова преобразовать (свернуть)
в двумерный. Я обойдусь без преобразований, но процедура от этого усложнится.
      Пузырьки будут путешествовать слева направо по строкам. Дойдя до конца
строки, они будут перепрыгивать в начало следующей, пока не уткнутся в
предыдущий пузырек.}

CONST   M=3; N=4;    {M - число строк в массиве, N - число столбцов}
TYPE    matritsa = array[1..M,1..N] of Word;
CONST  a          : matritsa =  ((2,6,4,2),        {Исходный массив}
                                 (9,1,8,3),
                                 (5,7,3,8));
VAR    i,j :Word;

PROCEDURE puziryok_2 (var mass:matritsa; M,N:Word);
VAR  i,j, i1,j1, k :Word;  {i - строка, по которой плывет пузырек, j - столбец; i1-строка, в которой остановился
              предыдущий пузырек, j1 - соседний слева столбец, k - какой по счету пузырек плывет}
       c           :Integer;
LABEL   metka;
BEGIN
    i1:=M; j1:=N;
    for k:=1 to M*N-1 do begin      {запускаем пузырьков на 1 меньше, чем чисел}
        for i:=1 to M do            {пузырек перескакивает вниз на строку}
            for j:=1 to N do begin  {пузырек плывет направо}
                if NOT ((i<i1)OR(i=i1)AND(j<j1)) then goto metka; {если уткнулся в предыдущий пузырек, то
                                останавливайся}
                if j<>N then        {Обмен величинами между двумя соседними элементами в строке:}
                    if mass[i,j]<mass[i,j+1] then begin
                        c:=mass[i,j];
                        mass[i,j]:= mass[i,j+1];
                        mass[i,j+1]:=c
                    end{if};
                if (j=N)AND(i<>M) then {Обмен величинами между крайним правым элементом в одной строке и
                                крайним левым в следующей:}
                    if mass[i,j]<mass[i+1,1] then begin
                        c:=mass[i,j];
                        mass[i,j]:= mass[i+1,1];
                        mass[i+1,1]:=c
                    end{if}
            end{for j};
metka:if j1>1    then j1:=j1-1          {Вычисляем, где остановился пузырек}
                 else begin j1:=N; i1:=i1-1 end
    end{for k};
END;
BEGIN
    puziryok_2 (a,M,N);
```

```
      {Распечатываем отсортированный массив:}
   for i:=1 to M do begin
      for j:=1 to N do Write (a[i,j],' ');
      WriteLn
   end{for};
   ReadLn
END.
```

Задание 133

```
USES Graph, CRT, DOS;
VAR  Device, Mode                        : Integer;
     Chas1, Min1, Sec1, Sotki1,
     Chas2, Min2, Sec2, Sotki2, React    : Word;
BEGIN
   DirectVideo:=false;
   Device:=0;
   InitGraph(Device,   Mode,    'c:\tp\bgi');
   WriteLn('Увидев квадрат, нажимайте клавишу ввода');
   Randomize;
   Delay(1000+Random(20000));
   Rectangle(100,100,300,300);
   GetTime(Chas1,Min1,Sec1,Sotki1);
   ReadLn;
   GetTime(Chas2,Min2,Sec2,Sotki2);
   React := 100*(Sec2-Sec1) + (Sotki2-Sotki1);
   WriteLn('Время вашей реакции - ',React,' сотых долей секунды');
   ReadLn
END.
```

Задание 134

```
USES DOS;
VAR  God, Mes, Den, Den_Ned,  God1, Mes1, Den1, Den_Ned1        : Word;
     Den_Ned_Text                                               : String;
BEGIN
   GetDate(God, Mes, Den, Den_Ned);          {Запоминаем настоящую дату}
   WriteLn('Введите число, номер месяца и год');
   ReadLn (Den1, Mes1, God1);
   SetDate(God1, Mes1, Den1);                {Устанавливаем интересующую нас дату}
   GetDate(God1, Mes1, Den1, Den_Ned1);      {Узнаем номер дня недели интересующей нас даты}
   case Den_Ned1 of                          {По номеру получаем текст}
      0  :Den_Ned_Text:='воскресенье';
      1  :Den_Ned_Text:='понедельник';
      2  :Den_Ned_Text:='вторник';
      3  :Den_Ned_Text:='среда';
      4  :Den_Ned_Text:='четверг';
      5  :Den_Ned_Text:='пятница';
      6  :Den_Ned_Text:='суббота'
   end;
   WriteLn(Den1,   '.' ,Mes1,'.',God1,' - ', Den_Ned_Text);
   SetDate(God, Mes, Den);       {Восстанавливаем настоящую дату}
   ReadLn
END.
```

П6. Список литературы

Д.Б.Поляков, И.Ю.Круглов **«Программирование в среде Турбо Паскаль (версия 5.5)». Москва, Издательство МАИ, 1992 год. 576 страниц.**

Это основная книжка, которую я вам рекомендую после изучения моей для расширения и углубления знаний по Паскалю. Как вводный курс ее читать, конечно, нельзя. Она толстая и в ней много полезного материала. Ничего, что версия – 5.5. Разницу с 7.0 вы почувствуете очень не скоро. Я не знаю, может быть эта книга и переиздана с 1992 года, может быть и под другим названием. Но авторы – хорошие.

В.В.Фаронов **«Основы Турбо-Паскаля (6.0)». Москва, МВТУ-ФЕСТО ДИДАКТИК, 1992 год. 304 страницы.**

Е.А.Зуев **«Язык программирования Turbo Pascal 6.0» Москва, Унитех, 1992 год. 298 страниц.**

О.Е.Перминов **«Программирование на языке Паскаль» Москва, Радио и связь, 1988 год. 220 страниц.**

П7. Предметный указатель