

کتابخانه  
بایبلیوتیکا  
ПРОГРАММИСТА

В. К. ВЛАСОВ, Л. Н. КОРОЛЕВ  
А. Н. СОТНИКОВ

# Элементы информатики



БИБЛИОТЕЧКА  
ПРОГРАММИСТА

---

В. К. ВЛАСОВ, Л. Н. КОРОЛЕВ, А. Н. СОТНИКОВ

# ЭЛЕМЕНТЫ ИНФОРМАТИКИ

Под редакцией Л. Н. КОРОЛЕВА



МОСКВА «НАУКА»  
ГЛАВНАЯ РЕДАКЦИЯ  
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ  
1988

ББК 22.18

В58

УДК 519.6

Власов В. К., Королев Л. Н., Сотников А. Н. Элементы информатики / Под ред. Л. Н. Королева. — М.: Наука. Гл. ред. физ.-мат. лит., 1988. — 320 с. — ISBN 5-02-013769-3

Изложены основы информатики. Значительное внимание уделено понятию алгоритма и алгоритмизации. Рассматриваются приемы программирования на машинно-ориентированных языках и на языке бейсик. Даются примеры программирования с использованием микрокалькулятора.

Книга предназначена для широкого круга читателей.

Табл. 7, Ил. 91.

Рецензент доктор физико-математических наук *С. А. Абрамов.*

В  $\frac{1702070000-109}{053(02)-88}$  9-88

ISBN 5-02-013769-3

© Издательство «Наука».  
Главная редакция  
физико-математической  
литературы, 1988

## ОГЛАВЛЕНИЕ

Введение . . . . .	5
Глава 1. Алгоритмы и алгоритмизация . . . . .	15
§ 1.1. Понятие алгоритма . . . . .	15
§ 1.2. Алгоритмические системы . . . . .	20
§ 1.3. Понятие о математических моделях . . . . .	26
§ 1.4. Алгоритмизация . . . . .	31
Глава 2. О некоторых вычислительных алгоритмах . . . . .	40
§ 2.1. Понятие о приближенных значениях величин . . . . .	40
§ 2.2. Решение систем линейных алгебраических уравнений . . . . .	44
§ 2.3. Понятие об интерполировании многочленами . . . . .	50
§ 2.4. Метод наименьших квадратов . . . . .	55
§ 2.5. Приближенное решение уравнения $f(x)=0$ . . . . .	60
§ 2.6. Приближенные формулы для вычисления определенных интегралов . . . . .	68
§ 2.7. Вычисление некоторых элементарных функций . . . . .	75
Глава 3. ЭВМ — исполнитель алгоритмов . . . . .	78
§ 3.1. Структура ЭВМ . . . . .	78
§ 3.2. Представление команд в ЭВМ . . . . .	86
§ 3.3. Позиционные системы счисления . . . . .	92
§ 3.4. Перевод чисел из одной системы счисления в другую . . . . .	96
§ 3.5. Смешанные системы счисления . . . . .	102
§ 3.6. Представление чисел в ЭВМ . . . . .	107
Глава 4. Приемы программирования . . . . .	115
§ 4.1. Некоторые способы записи алгоритмов . . . . .	115
§ 4.2. Понятие о символьном кодировании . . . . .	127
§ 4.3. Описание системы команд машины . . . . .	132
§ 4.4. Программирование вычислений по формулам . . . . .	134
§ 4.5. Программирование разветвляющихся процессов . . . . .	136
§ 4.6. Программирование циклов . . . . .	145
§ 4.7. Программирование вложенных циклов . . . . .	156
§ 4.8. Подпрограммы . . . . .	158

Глава 5. Программирование на бейсике . . . . .	165
§ 5.1. Понятие о языках высокого уровня . . . . .	165
§ 5.2. Основные символы языка бейсик. Выражения . . . .	173
§ 5.3. Операторы языка бейсик . . . . .	177
§ 5.4. Диалог «Человек — ЭВМ» . . . . .	216
Глава 6. Обработка текстовой информации . . . . .	224
Глава 7. Машинная графика . . . . .	247
Глава 8. Вычислительные машины и их математическое обеспечение . . . . .	262
Приложение. Вычисления с помощью микрокалькуляторов	271
§ П1. Общие сведения о микрокалькуляторах . . . . .	271
§ П2. Некоторые приемы решения задачи на простейших микрокалькуляторах . . . . .	281
§ П3. Программируемые микрокалькуляторы . . . . .	293
§ П4. Основные приемы программирования ПМК . . . . .	302
Таблица основных технических характеристик ПЭВМ советского производства . . . . .	311
Предметный указатель . . . . .	315

## ВВЕДЕНИЕ

Во все времена своего существования человек нуждался в инструментах для счета. В первобытные времена таким инструментом у человека была его собственная рука. (Пальцевый счет использовался человеком и на более высоких ступенях развития цивилизации — в Древней Греции и в Древнем Риме.)

Затем человек стал использовать для счета самые примитивные устройства — сначала это были деревянные палочки с зарубками — бирки (первое упоминание о них на барельефе фараона Сети I относится примерно к 1350 г. до н. э.); китайцы, персы, индийцы пользовались для представления чисел и счета ремнями и веревками с узлами. Записывать числа и вести счет на бумаге было невозможно по той простой причине, что ее еще не существовало, пергамент же был слишком дорог.

Острая необходимость в более совершенном инструменте для записи чисел и счета привела к появлению *абака* (в переводе на русский язык — дощечка, покрытая пылью). По свидетельству Геродота египтяне им пользовались уже в V в. до н. э. По существу, этот инструмент похож на обычные русские счеты.

В XVII в. были изобретены логарифмы и сразу вслед за этим был создан новый счетный инструмент — логарифмическая линейка. Примерно в то же время на свет появились арифметические машины Шиккарда, Паскаля, Лейбница. Это уже были механические устройства, использующие многочисленные шестерни, колеса, зубчатые рейки и т. п. Они умели «запоминать» числа и выполнять элементарные арифметические операции. В действие они приводились человеческой рукой, причем зачастую для этого требовались недюжинные усилия.

Особую роль в развитии вычислительной техники сыграли работы выдающегося английского ученого Чарльза Беббиджа. В начале XIX в. он предложил идею создания «разностной» машины, которая предназначалась для вычисления значений многочленов без вмешательства человека в процесс счета, т. е. машина должна была считать *автоматически*. И такая машина была им создана. Но Беббидж мечтал об универсальной машине, на которой можно было бы

решать произвольные вычислительные задачи. Всю жизнь посвятил Беббидж разработке такой машины, которую сам он назвал «аналитической». Беббидж составил подробную схему машины, выполнил огромное количество чертежей отдельных узлов, воплотил в металле некоторые ее части. Но полностью создать машину он, конечно же, не мог. Идеи Беббиджа намного опередили свое время, уровень развития науки и техники того времени не позволил ему до конца решить поставленную задачу. И хотя с тех пор прошло полтора столетия, все современные вычислительные машины по своей структуре в значительной степени похожи на аналитическую машину Беббиджа.

Так, по проекту Беббиджа машина должна была обладать запоминающим устройством для хранения чисел (он называл его «складом»), устройством, способным выполнять арифметические операции (по Беббиджу — «мельница»), устройством, управляющим последовательностью действий машины, устройством ввода и вывода информации, т. е. практически всеми основными элементами современных вычислительных машин.

Разумеется, машина Беббиджа, как и предшествующие арифметические машины, задумывалась как чисто механическое устройство. По проекту автора, она должна была выполнять сложение за одну секунду, умножение и деление примерно за одну минуту. Вводить информацию и управлять вычислительным процессом Беббидж предлагал с помощью перфокарт, т. е. картонных карточек с пробитыми на них отверстиями. Эту идею он заимствовал у француза Жаккара, который в начале XIX в. изобрел способ управления ткацкими станками с помощью перфокарт.

Беббидж высказал исключительно важную мысль о возможности изменения хода вычислений в зависимости от промежуточных результатов. Такое свойство машины уже позволяло решать на ней весьма сложные задачи. Однако для того чтобы заставить машину выполнять необходимые вычисления, нужно было составить для нее программу, т. е. последовательность инструкций (команд) для всех ее устройств. И первые такие программы для аналитической машины Беббиджа были составлены талантливым математиком Адой Лавлейс (дочерью поэта Байрона), которую по праву можно назвать первым программистом в истории человечества.

Конечно же, эти программы существовали только на бумаге, так как сама машина так и не была построена и не было возможности выполнить эти программы. Лишь примерно через сто лет идеи Беббиджа начали реально воплощаться в жизнь.

В 1937 г. болгарин А. Атанасов приступил к созданию электронной вычислительной машины, предназначенной для решения задач математической физики. Одновременно проект большой релейной машины предложил сотрудник Гарвардского университета в США Г. Айткен. Машина была построена в 1944 г. и называлась «Марк-1».

Но это была электромеханическая машина, а в 1945 г. в США появилась первая электронная вычислительная машина (ЭВМ) — ЭНИАК. Это была уже ламповая машина на электронных реле. Однако она имела существенный недостаток — для того чтобы изменить режим работы, нужно было каждый раз вручную перекоммутировать схему.

Следующий важный шаг в развитии вычислительных машин сделал знаменитый американский математик Джон фон Нейман. Он предложил вводить в память машины наряду с исходными числовыми данными также и саму программу. Такой принцип был назван принципом хранимой программы. Теперь уже можно было рассматривать программу тоже как объект для воздействия, т. е. ее уже можно было изменять прямо в процессе вычислений. Программа могла теперь сама себя перерабатывать и изменяться по мере необходимости в зависимости от результатов вычислений.

В Советском Союзе первая электронная вычислительная машина была создана под руководством академика С. А. Лебедева в 1951 г. Называлась она МЭСМ — малая электронная счетная машина. В 1952 г. была построена самая мощная в то время в Европе машина — БЭСМ-1. А в 1953 г. появились первые серийные советские ЭВМ — «Стрела» и М-2, причем эти машины по меркам того времени обладали широкими вычислительными возможностями и высоким быстродействием.

Далее развитие вычислительной техники пошло семимильными шагами. Если в 1952 — 1953 гг. электронных машин в мире было несколько десятков, то в 1965 г. — около 40 тыс., в 1970 г. — более 100 тыс., в 1977 г. — около 400 тыс. и т. д. В настоящее время самостоятельно используемых ЭВМ в мире насчитывается уже несколько миллионов, не считая микроЭВМ, встроенных в различные приборы и автоматы.

Но не только росло количество ЭВМ, еще быстрее они совершенствовались.

Электронные вычислительные машины (или, как теперь их часто называют, компьютеры) принято делить на поколения. В основу градации обычно кладут элементную базу, на которой строятся машины, а также их возможности, область применения и т. д. Деление это весьма условно, ибо случается, что машина, построенная на элементной базе одного поколения, по структурным особенностям и возможностям относится к машинам другого поколения. (Так, например, серийная советская машина БЭСМ-6 по элементной базе относится к ЭВМ второго поколения, а по своей архитектуре близка к машинам третьего поколения.)

К настоящему времени обычно насчитывают четыре или даже пять поколений ЭВМ.

Машины первого поколения строились на электронных лампах. Их быстродействие достигало 20 тыс. операций в секунду. Программы писались на языке машины. Причем каждая машина имела свой,



присущий только ей язык, и если программа была написана на этом языке, то на другой ЭВМ ее уже использовать было нельзя. Объем памяти машин был весьма небольшим. Программист сам распределял оперативную память под программу, исходные данные, полученные результаты.

В ЭВМ второго поколения на смену электронным лампам пришли полупроводниковые приборы — транзисторы, которые позволили довести быстродействие до нескольких сотен тысяч операций в секунду, одновременно значительно увеличив надежность машин. Произошел переход от написания программ на языке машины к написанию их на алгоритмических языках высокого уровня. Запись программы на алгоритмическом языке использует некоторые слова и стандартную математическую символику. Такая запись значительно проще и нагляднее программы, написанной на языке машины. Программист, составляющий такую программу, может и вовсе не знать языка машины. Перевод программы с алгоритмического языка на язык машины осуществляется самой машиной с помощью специальной программы-транслятора.

В конце 60-х гг. наступил следующий этап в развитии ЭВМ — появились машины третьего поколения. Их рождение связано с ростом возможностей технологии полупроводников. На одном кремниевом кристалле, а точнее кристаллической пластинке, удалось создать электронную схему, содержащую сотни и даже тысячи простейших элементов. Такую схему называли *интегральной*. Она стала эквивалентной целому блоку ЭВМ. Машины, построенные на интегральных схемах, имеют быстродействие порядка миллиона операций в секунду. Использование интегральных схем повысило надежность, уменьшило размеры ЭВМ.

В машинах третьего поколения стал применяться принцип параллелизма работ. ЭВМ стала одновременно обрабатывать несколько программ. Общение с ЭВМ осуществляется сразу с нескольких терминалов, т. е. удаленных от ЭВМ пультов, снабженных клавиатурой и экраном. Каждый работающий за таким пультом решает свою задачу и не ощущает, что на этой же ЭВМ одновременно с ним другие пользователи решают свои задачи.

Последнее достижение на пути миниатюризации — микроминиатюрные интегральные схемы, когда на крошечной кремниевой пластине содержатся уже десятки тысяч электронных компонент. Это так называемые *большие интегральные схемы* (БИС). Более того, созданы сверхбольшие интегральные схемы (СБИС), насчитывающие на поверхности кристалла сотни тысяч электронных компонент. БИС и СБИС стали элементной базой ЭВМ четвертого поколения. Новое поколение значительно превосходит старые ЭВМ по быстродействию, достигая десятков и даже сотен миллионов операций в секунду. Если для машин первых поколений требовались большие помещения, слож-

ные системы охлаждения и вентиляции, то современные машины значительно меньше по размерам и менее прихотливы к условиям эксплуатации. Кстати, отметим, что их миниатюризация позволяет не только экономить помещение — она сказывается и на быстродействии. В самом деле, сейчас скорости работы различных устройств ЭВМ настолько высоки, что начинает играть роль время прохождения сигнала от одного устройства к другому, хотя эти скорости и сопоставимы со скоростью света. Уменьшая длину пути прохождения сигнала, удастся повысить скорость работы ЭВМ.

Современные ЭВМ обладают значительно большими возможностями, чем старые машины, — на них, как уже говорилось, можно одновременно решать несколько задач, к ним можно обращаться с удаленных выносных пультов (терминалов). Кроме того, они значительно надежнее и, что немаловажно, стоимость решения задач на них значительно ниже, чем на старых ЭВМ. Очень часто машины используют не поодиночке, их объединяют в мощные вычислительные комплексы и сети. Тогда их возможности резко возрастают.

Широко обсуждается вопрос о создании ЭВМ пятого поколения, которые обеспечат новый уровень взаимодействия человека с ЭВМ. Отличительной их особенностью должен стать повышенный уровень интеллекта.

Разработчики стремятся к тому, чтобы с машинами можно было общаться на естественном языке, чтобы с их помощью накапливались знания, чтобы ЭВМ могли обучать и обучаться.

В настоящее время очень широкое распространение получили *персональные ЭВМ*. Эти малогабаритные машины, сравнительно дешевы, неприхотливы, просты в эксплуатации, но обладают довольно большими возможностями при решении широкого круга задач. Размещать такие ЭВМ можно на обычном письменном столе.

Миниатюризация элементной базы привела к созданию *микропроцессоров*. Микропроцессоры могут быть вмонтированы в наручные часы и теперь на руке у нас не только собственнo часы, но и календарь, и калькулятор. Микропроцессоры управляют работой стиральной машины-автомата, запоминают номера телефонов в телефонном аппарате, без микропроцессоров трудно себе представить современную радиоаппаратуру.

Менялись машины, менялись поколения ЭВМ, а вместе с ними менялся и характер их применения. Если сначала они создавались и использовались в основном для решения разного рода вычислительных задач, то в дальнейшем сфера их применения необычайно расширилась. Теперь это уже и задачи обработки результатов физических экспериментов, и моделирование различных объектов естествознания, и задачи управления как народным хозяйством в целом, так и отдельными предприятиями и отраслями производства.

Трудно переоценить роль, которую играют ЭВМ в геологии и

медицине, в биологии и лингвистике, в химии и экономике, в исследовании космоса и ядерной физике. Машины помогают конструкторам в создании самолетов и космических кораблей, архитекторам в проектировании зданий. Машины управляют станками, составляют расписание движения поездов и самолетов, переводят тексты с одного языка на другой, управляют полетом космических аппаратов, ставят диагнозы болезней.

С помощью ЭВМ удается решать задачи, решение которых раньше считалось невозможным из-за огромного объема вычислений, или даже если и было возможным, то терялся смысл из-за слишком большой продолжительности решения задачи. Эксперименты и подсчеты, требовавшие раньше тысяч часов рутинной работы сотен людей, теперь выполняют за несколько минут и с высокой точностью.

Скажем, решение задачи краткосрочного прогноза погоды требует большого количества сложных математических расчетов. Ясно, что не слишком велика была бы польза, если бы мы выполнили все необходимые расчеты для прогноза погоды на завтра лишь послезавтра. Машина все подсчитывает точно и в срок. Если же прогноз иногда и не оправдывается, то не по вине машины, а из-за того, что неудачно составлены те уравнения, которые ей пришлось решать, или, как говорят, неудачно построена математическая модель изучаемого явления — в данном случае модель погоды. Или же неточны исходные величины, заданные в этой модели.

Все более широкое применение находят ЭВМ и как средства обучения. Машина может быть самым внимательным и терпеливым преподавателем. Причем занятия она ведет с каждым [обучаемым индивидуально, фиксирует любой пробел в знаниях и способствует глубокому и прочному усвоению изучаемой дисциплины.

Очень часто говорят, что ЭВМ осуществляет обработку данных, говорят об алгоритмах обработки данных, об исходных данных, о кодировании данных и т. д. В том же смысле употребляют термин «обработка информации», поэтому важно [уяснить себе, что под этим понимается.

Слово «информация» переводится на русский язык как сведения. Информацию любой природы, зафиксированную [каким-либо образом, мы и будем называть *данными*. Информация может быть зафиксирована в книгах с помощью букв, знаков, картинок и цифр или каким-либо другим способом, например в виде записей на магнитных лентах, на фотокдрах и, пока еще не очень хорошо известным нам способом, в мозге человека. [Физическую среду, на которой (или внутри которой), зафиксирована информация, называют *носителем* информации. Носителем информации может быть бумага, перфолента, перфокарта, магнитный диск и многое другое. Современная техника предлагает все новые и новые носители информации, основанные на использовании электрических, магнитных и оптических свойств раз-

личных материалов и даже свойств отдельных молекул. Существенно, чтобы носитель информации позволял читать зафиксированную информацию. То, что написано в книгах, может читать человек, то, что зафиксировано с помощью электрических сигналов, магнитных записей может «читать» соответствующее техническое устройство.

Данные, физически зафиксированные на любом носителе информации, с той или иной степенью точности можно представлять себе в виде последовательности символов.

В самом деле, всякая книга может быть рассмотрена как последовательность символов, обозначающих буквы, цифры и другие знаки. Газетная фотография представляет собой последовательность чередующихся черных и светлых точек. Чтобы убедиться в этом, рассмотрите газетную фотографию через увеличительное стекло. Вообще, любую фотографию с некоторой степенью приближения можно рассматривать как последовательность точек разного цвета или степени затемнения. Любой график также можно рассматривать как последовательность координат, выраженных числами, т. е. как некоторую последовательность цифр.

Очень важно понять, что все сведения об объектах реального мира, о явлениях природы, о результатах размышлений, о научных законах и выводах человечество научилось фиксировать в виде последовательностей символов, в виде данных. Вычислительные машины прямо приспособлены именно для обработки символьных последовательностей, для их преобразования из одной формы в другую, для выполнения арифметических и логических операций над такими последовательностями. Неудивительно поэтому, что в лексикон, связанный с функционированием ЭВМ, вошли такие выражения, как «обработка слов», «анализ предложений», «обработка строк». И еще важно отметить, что человечеству понадобилось не так уж много различных символов, чтобы написать миллионы томов книг. Например, в текстах книг на русском языке использовано 33 буквы, 10 цифр и около десятка других различных знаков, т. е. с помощью всего нескольких десятков различных символов удается зафиксировать колоссальные объемы информации. Как это делается, мы с вами хорошо знаем — из букв формируются слова, из слов — предложения, из предложений — осмысленный текст статьи, книги и т. д. Те же данные можно представить, используя меньшее число букв. Например, в латинском алфавите букв меньше, чем в русском (их 26). Но нам известно, что все написанное с использованием русского алфавита можно перевести на любой другой язык без потери сведений, содержащихся в исходном тексте.

Более того, оказывается, достаточно всего двух различных символов, т. е. алфавита, состоящего всего из двух букв, чтобы с их помощью зафиксировать все имеющиеся в распоряжении человечества сведения, всю информацию. Мы знаем, например, что телеграфист

с помощью азбуки Морзе, используя всего два символа — точку и тире, может передать любой текст. Этот факт и лежит в основе использования в ЭВМ алфавита, состоящего всего из двух символов.

Приведенные рассуждения имели своей целью объяснить читателю, почему ЭВМ, первоначально предназначенные для быстрого автоматического выполнения вычислений, оказались универсальным инструментом обработки данных самого различного происхождения и характера. Они способны автоматически выполнять алгоритмы преобразования символьных последовательностей, и этого оказалось достаточно, чтобы использовать их для выполнения любых алгоритмов и решать задачи, не имеющие отношения к расчетам: задачи типа анализа текстов, формирования чертежей, сложные логические задачи.

Укоренившееся название «электронная вычислительная машина» отражает лишь очень небольшую часть возможностей ЭВМ по обработке данных.

В основе дальнейшего развития научно-технического прогресса лежит широкое, повсеместное внедрение в промышленное производство и другие сферы деятельности человеческого общества последних достижений науки, техники, технологии. Интенсификация производства, научной деятельности, образования немыслима без автоматизации, без внедрения автоматизированных линий, гибко перестраиваемых производств, без автоматизации управленческой деятельности. В настоящее время основным звеном, по существу, всех систем автоматизации является цифровая автоматика и вычислительная техника. Партия и правительство уделяют развитию этой отрасли знаний и производства самое серьезное внимание. Принят перспективный план развития вычислительной техники и ее повсеместного внедрения в народное хозяйство. В ближайшие годы с вычислительной техникой и ее применением будут связаны тем или иным образом все члены общества, вовлеченные в общественное производство.

Процесс широкого внедрения вычислительной техники иногда называют процессом информатизации общества, а все, что связано с применением ЭВМ, их разработкой, созданием программ для них, называют *информатикой*. В каком-то смысле понятие «информатика» подменяет собой то, что называется кибернетикой. Как мы знаем, *кибернетика* определяется как наука об общих законах управления в живой и неживой природе. На современном этапе управление в многообразных сферах «неживой» природы (в производстве, науке, технике) осуществляется с помощью ЭВМ. В этом смысле информатику можно назвать частью кибернетики.

В то же время широкое применение вычислительной техники для научных расчетов, научного прогнозирования, анализа результатов экспериментов, учета и контроля, т. е. в сферах, прямо не связанных с управлением, позволяет считать, что информатика охватывает дру-

гой, более широкий класс проблем и решает более разнообразные задачи по сравнению с кибернетикой в классическом ее понимании.

Если же отвлечься от самых общих определений, то под *информатикой* понимают науку, связанную:

1) с разработкой вычислительных машин и систем, с технологией их создания;

2) с разработкой математических моделей естествознания и общественных явлений с целью их строгой формализации;

3) с обработкой данных, созданием численных и логических методов решения задач, сформулированных на этапе построения математической модели;

4) с разработкой алгоритмов решения задач управления, расчета и анализа математических моделей;

5) с программированием алгоритмов, созданием программного обеспечения ЭВМ.

Огромный вклад в развитие информатики в ее современном понимании внесли выдающиеся ученые и организаторы науки академики С. А. Лебедев, М. А. Лаврентьев, М. В. Келдыш, В. М. Глушков и др.

Интенсификация производственной деятельности, ускорение научно-технического прогресса невозможны без широкого массового внедрения средств вычислительной техники во все сферы деятельности нашего общества. Этим определена необходимость изучения основ информатики и вычислительной техники на всех уровнях образования, начиная со средней школы, техникума, ПТУ и т. д. Возникает естественный вопрос — посильная ли это задача — достижение в нашей стране всеобщей компьютерной грамотности. Тут напрашивается аналогия, которую подметил академик Б. Сендов (НРБ). Он сказал, что нынешняя ситуация напоминает эпоху начала телефонизации: когда только появилась телефонная связь, в городах было всего несколько десятков или сотен телефонных аппаратов. Чтобы позвонить по телефону, надо было снять трубку, соединиться с «телефонной барышней», назвать ей номер или даже просто имя абонента (не так уж много их было и нетрудно было запомнить их все). «Барышняя» с помощью проводков со штеккерами на концах соединяла с абонентом, и происходил разговор. Люди с ужасом думали о том, что будет, если количество телефонов в городах резко возрастет. Сколько же тогда понадобится «барышень», чтобы обеспечить бесперебойную связь?! Прошло время, в больших городах теперь сотни тысяч и даже миллионы телефонных аппаратов. И каждый из нас сам выполняет роль телефонной барышни. Достаточно снять трубку и набрать нужный номер, как тут же произойдет соединение. Правда при этом нам помогает АТС (автоматическая телефонная станция) — устройство исключительно сложное. И именно это самое устройство дает нам возможность с такой легкостью соединиться с любым аба-

нением города, а также без вмешательства телефонистки, набрав код другого города в нашей стране или даже за рубежом, мы можем разговаривать с самыми удаленными точками планеты.

Аналогичная ситуация происходит и с использованием вычислительной техники. На заре создания ЭВМ, в эпоху машин первого поколения, писать программы умели лишь профессионалы-программисты. Математики, физики, биологи—все, кому надо было решать задачи на машине, обращались за помощью к этим профессионалам, и те составляли соответствующие программы.

Шло время, появились алгоритмические языки высокого уровня, близкие к естественным языкам общения. Уже значительно более широкий круг специалистов мог самостоятельно работать на ЭВМ. И чем более совершенствовалась вычислительная техника, чем более она усложнялась, тем проще становилось общение с ней, т. е. все происходит по той же схеме, что и с телефонизацией. И для того чтобы использовать вычислительную технику, решать на ней различные задачи, вовсе не обязательно глубоко и подробно знать ее устройство. Ведь мы умеем пользоваться телевизором, технически весьма сложным прибором, хотя далеко не все знают тонкости его конструкции. Современная ЭВМ—очень сложное устройство, аппаратные средства которой в сочетании с программным обеспечением помогают просто и эффективно решать с ее помощью разнообразные задачи. Общение с ЭВМ становится настолько простым, что ему уже можно обучать каждого, кто этого пожелает, т. е. такому общению можно научить уже в школе. Но обучая основам информатики, нужно ставить себе целью не только и даже не столько научить обучаемого написанию программ для ЭВМ, сколько развить в нем способность к алгоритмическому мышлению, привить умение строить и анализировать алгоритмы решения различных задач.

Умение выделить в процессах, явлениях природы, производственной деятельности людей «алгоритмическую» сторону эквивалентно пониманию механизмов этих явлений и процессов, отчетливому восприятию их причинно-следственной связи.

В предлагаемой книге, рассчитанной на широкий круг читателей, разумеется, нельзя охватить все направления, входящие в состав информатики. Главное внимание в ней авторы сосредоточили на изложении понятий алгоритмизации и программирования. При этом авторы стремились к элементарному изложению, понимая под этим не столько упрощение, а главным образом доходчивость изложения подчас не очень простых вещей, связанных с данным предметом.

## ГЛАВА I

# АЛГОРИТМЫ И АЛГОРИТМИЗАЦИЯ

### § 1.1. Понятие алгоритма

В своей повседневной деятельности нам постоянно приходится сталкиваться с разнообразными правилами, предписывающими последовательность действий, цель которых состоит в достижении некоторого необходимого результата. [Подобные правила очень многочисленны. Например, мы обязаны следовать вполне определенной системе правил, чтобы позвонить по телефону-автомату (опустить монету, снять трубку, набрать [номер и т. д.) или пройти через турникет в метро, вычислить произведение двух многозначных чисел или найти корни квадратного уравнения. Нужно выполнить определенную последовательность действий, чтобы сварить суп или подняться в лифте на нужный этаж, с помощью циркуля и линейки разделить отрезок пополам или вычислить сумму бесконечно убывающей геометрической прогрессии, приготовить лекарство или связать на спицах свитер. Примеры такого рода можно продолжать неограниченно. Их часто называют *алгоритмами*.

В особо четких формулировках предстают перед нами алгоритмы, связанные с правилами выполнения арифметических действий и геометрических построений. Не будет большой ошибкой сказать, что элементарная (и не элементарная) математика в значительной степени сводится к нахождению правил решения задач.

Название «алгоритм» связано с именем выдающегося математика древности Мухаммеда бен-муса аль-Хорезми (IX в. н. э.). Он изложил общие правила выполнения действий над числами, представленными в десятичной форме, которыми мы пользуемся до сих пор. Существенным было то, что эти правила могли быть применены к любым числам.

До этой поры существовало очень много приемов выполнения арифметических операций, в которых учитывались те или иные особенности конкретных чисел, над которыми проводились эти операции. Например, если один из сомножителей представляет число, состоя-



щее из одних девяток, то к другому сомножителю нужно дописать число нулей, равное числу девяток в первом сомножителе, и затем из получившегося числа вычесть второй сомножитель. Тем самым мы получим нужный результат. Например, если число 999 нужно умножить на 5, мы к цифре 5 добавим три нуля (так как в первом сомножителе три девятки), получим 5000 и затем вычтем второй сомножитель, т. е. число 5. Результатом будет число 4995, которое и является произведением чисел 999 и 5.

Мы и сейчас пользуемся многими приемами быстрого устного счета. Например, чтобы умножить любое число на 5, надо приписать к нему 0 и разделить полученное число пополам. Или, чтобы умножить число на 25, достаточно умножить это число на 100, т. е. приписать к нему два нуля, и разделить результат на 4. Однако такого рода приемы можно применять только в тех случаях, когда хотя бы один из сомножителей имеет специальный вид (все девятки, 5, 25 и т. д.).

Аль-Хорезми предложил правила, пригодные во всех случаях и одинаковые для любых пар чисел. Сторонников методики аль-Хорезми начали называть алгоритмками, а под словом «алгоритм» стали понимать систему правил, обладающих некоторыми определенными свойствами.

Действия согласно той или иной инструкции, являющейся алгоритмом, для получения определенного результата предполагают наличие некоторых исходных данных.

Например, исходными данными при выполнении арифметической операции над двумя числами является эта пара чисел, а результатом — одно число.

Первым свойством алгоритма является *дискретный*, т. е. пошаговый характер определяемого им процесса.

Другим важнейшим свойством алгоритмов является *массовость*. Смысл данного понятия заключается в том, что существует некоторое множество объектов, которые могут служить исходными данными для рассматриваемого алгоритма. Например, для алгоритмов выполнения арифметических операций — сложения, вычитания, умножения и деления — такими данными являются все действительные числа. Когда складываем, например, два числа 27 и 35, то пользуемся алгоритмом сложения столбиком:

$$\begin{array}{r} 27 \\ + 35 \\ \hline 62 \end{array}$$

Здесь мы не просто складываем цифры в соответствующих разрядах, но и в зависимости от результата сложения осуществляем перенос единиц в более старшие разряды. Заметим, кстати, что когда мы складываем однозначные числа, т. е. попросту говоря, цифры, то

пользуемся таблицей сложения:

$$0+0=0; 0+1=1; 1+0=1; 1+1=2; \dots \\ 9+8=17; 9+9=18.$$

Разумеется, этой таблицей мы пользуемся и применяя алгоритм сложения многозначных чисел.

То же самое можно сказать и об умножении двух чисел. Перемножая цифры, мы пользуемся лишь хорошо известной нам таблицей умножения. Если же мы перемножаем многозначные числа, то тут приходится применять алгоритм умножения столбиком:

$$\begin{array}{r} \times 27 \\ 35 \\ \hline 135 \\ 81 \\ \hline 945 \end{array}$$

Здесь мы пользуемся таблицей умножения, есть и запоминание результата умножения цифр, и перенос в старшие разряды, и алгоритм сложения столбиком. Заметим, что алгоритм умножения столбиком включает в себя как составную часть алгоритм сложения.

Подчеркнем еще раз, что алгоритм умножения остается одним и тем же для любых пар чисел. Иными словами, исходными данными для этого алгоритма могут быть любые пары чисел. Конечно, такой способ умножения чисел не всегда самый рациональный. Так, при применении этого алгоритма в том случае, когда один из сомножителей состоит только из девяток, мы затратим больше усилий по сравнению с приемом быстрого счета. Но смысл массовости алгоритма состоит как раз в том, что он одинаково пригоден для всех случаев, требует лишь механического выполнения цепочки некоторых простых действий и при этом нет нужды в затратах творческой энергии.

В свое время противники методов аль-Хорезми именно за это пренебрежительно относились к «алгоритмикам», снявшим покров тайственности с искусства выполнять арифметические операции.

Приведем пример еще одного алгоритма, а именно алгоритма деления отрезка пополам с помощью циркуля и линейки. Исходными данными здесь уже являются не числа, а заданный отрезок. Действия — построение с помощью циркуля и линейки.

Для того чтобы разделить отрезок пополам, возьмем раствор циркуля, равный длине отрезка, и из концов отрезка опишем этим радиусом окружности (рис. 1.1). Через точки пересечения окружностей проведем прямую. Эта прямая пересечет заданный отрезок в точке, которая, как известно, и является серединой данного отрезка.

Подчеркнем, что приведенная последовательность действий обладает свойством массовости, поскольку ее можно применять к любому отрезку для отыскания его середины.

Читатель, конечно же, заметил, что для решения поставленной задачи вовсе не обязательно, чтобы радиус описываемых окружностей

был равен длине заданного отрезка — достаточно, чтобы эти радиусы были одинаковыми и большими, чем половина длины отрезка. Наше ограничение преследовало цель сделать алгоритм четким и однозначным.

Остановимся еще на одной важной особенности, присущей каждому алгоритму. Предполагается, что алгоритм *понятен* для исполни-

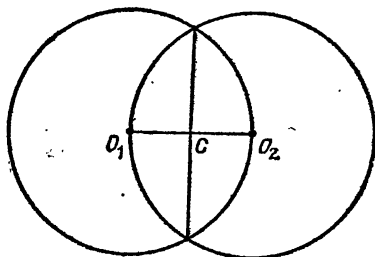


Рис. 1.1

теля, т. е. исполнитель алгоритма знает, как его выполнять. При этом исполнитель алгоритма, выполняя его, действует «механически». Очевидно, что формулировка алгоритма должна быть настолько точна и однозначна, чтобы могла полностью определять все действия исполнителя.

Анализ приведенных выше алгоритмов показывает, что если

применять алгоритмы повторно к одним и тем же исходным данным, то мы всегда будем получать один и тот же результат. Например, если применять рассмотренный ранее алгоритм деления отрезка пополам к одному и тому же отрезку, то каждый раз будем получать одну и ту же точку.

И если при этом каждый раз сравнивать результаты, полученные после соответствующих шагов алгоритмического процесса, то окажется, что при одних и тех же исходных данных эти результаты всегда будут одинаковыми. Таким образом, можно говорить об *определенности* и *однозначности* алгоритмов.

Теперь можно более точно определить алгоритм как систему правил, сформулированную на языке, понятном исполнителю, и определяющую цепочку действий, в результате выполнения которых мы приходим от исходных данных к искомому результату. Такая цепочка действий называется алгоритмическим процессом, а каждое действие — его шагом. Число шагов для достижения результата обязательно должно быть конечным. Кроме того, алгоритм должен обладать свойствами массовости, определенности и однозначности.

Механический характер алгоритма, его определенность и однозначность позволяют в качестве исполнителя рассматривать и специальные машины-автоматы.

Неправильно было бы думать, что для любой задачи существует лишь один алгоритм ее решения. Пусть, например, надо сосчитать количество зрителей на трибунах стадиона. Можно сосчитать количество зрителей в каждом секторе и потом их сложить. А можно посчитать количество зрителей, сидящих в первом ряду по всему периметру стадиона, сложить с количеством зрителей, сидящих во

втором ряду, затем в третьем и т. д. Результат в обоих случаях, естественно, должен получиться одинаковым.

Однако далеко не всегда даже для внешне кажущейся легкой задачи удастся найти хоть какой-нибудь алгоритм ее решения.

Так, на протяжении тысячелетий математики пытались решить задачу о квадратуре круга: с помощью циркуля и линейки построить квадрат, равновеликий кругу с заданным радиусом  $r$ , т. е. так, чтобы  $\pi r^2 = x^2$ , где  $x$  — сторона искомого квадрата. Лишь в XIX в. усилиями выдающихся математиков Лежандра и Линдемана была строго доказана неразрешимость этой задачи. В самом деле, из условия задачи следует, что  $x = r\sqrt{\pi}$ . Поэтому надо осуществить построение так, чтобы получить отрезок, длина которого в  $\sqrt{\pi}$  раз больше длины данного отрезка. Оказывается «умножить» графически с помощью циркуля и линейки отрезок на число можно лишь тогда, когда это число является корнем алгебраического уравнения с целыми коэффициентами. Лежандр установил, что  $\pi$  — число иррациональное (т. е. непредставимо в виде  $p/q$ , где  $p$  и  $q$  — взаимно простые числа), а Линдеман — что оно (а значит, и  $\sqrt{\pi}$ ), к тому же, и трансцендентное, т. е. не удовлетворяет никакому алгебраическому уравнению с целыми коэффициентами.

Столь же известна другая задача древнегреческой математики — задача о трисекции угла, т. е. о делении угла на три равные части с помощью циркуля и линейки. Как и в случае задачи о квадратуре круга, неразрешимость этой задачи была доказана лишь в XIX в. и тоже алгебраическими методами.

Упомянем, наконец, еще об одной замечательной задаче древности — удвоение куба: требуется построить куб, объем которого был бы вдвое больше объема заданного куба. Лишь в 1837 г. было доказано, что не существует алгоритма, который бы позволил с помощью циркуля и линейки решить поставленную задачу.

Заметим, что во всех трех знаменитых задачах древности для их решения разрешалось пользоваться только циркулем и линейкой. И оказалось, что этих средств недостаточно, чтобы решить каждую из поставленных задач. Однако отсюда вовсе не следует, что эти задачи нельзя решить, применяя какие-либо другие средства. Те же древние греки находили для этого хитроумные способы.

В указанных примерах речь шла о задачах, для которых (и это строго доказано) не существует алгоритмов решения (т. е. задачи неразрешимы) в рамках фиксированного набора допустимых действий, в данном случае — построения только с помощью циркуля и незамеченной линейки.

Есть также множество проблем, о которых мы и сейчас не можем сказать, разрешимы они или нет, и если разрешимы, то каким образом.

Так известна гипотеза о том, что любое четное число представимо в виде суммы двух простых чисел. Но утверждение это пока не доказано, и не ясно, удастся ли его когда-либо доказать или опровергнуть.

Итак, из сказанного выше вытекает, что для некоторых задач существует несколько алгоритмов их решения, для некоторых таких алгоритмов вообще не существует, и, наконец, есть задачи, для которых мы не знаем, существуют или нет алгоритмы их решения.

## § 1.2. Алгоритмические системы

Когда речь идет о построении алгоритма решения какой-либо конкретной задачи, то мы явно или неявно предполагаем известными те объекты, которые будут исходными данными для нашего алгоритма. Например, в задаче деления отрезка пополам в качестве таких объектов выступают отрезки произвольной длины. При построении алгоритма нахождения корней квадратного уравнения такими объектами являются тройки любых чисел, определяющие коэффициенты уравнения  $ax^2 + bx + c = 0$ .

Второе, что мы предполагаем известным, — это те действия, с помощью которых строятся шаги алгоритма. Например, в той же задаче о делении отрезка мы разрешаем использовать только действия с циркулем и линейкой, а в задаче решения уравнения разрешается выполнять действия сложения, вычитания, умножения, деления и извлечения квадратного корня. Иными словами, мы предполагаем известными все возможности исполнителя алгоритма, т. е. знаем, что умеет делать исполнитель (ведь алгоритм обязательно адресован какому-то конкретному исполнителю).

Мы также должны знать, как формулировать шаги заданий, чтобы их понял исполнитель, т. е. знать, какой язык понятен исполнителю алгоритма. Наконец, при построении алгоритма решения какой-либо конкретной задачи мы должны знать, что собой будет представлять результат: число, точку на прямой и т. д., т. е. мы должны знать множество объектов, к которым принадлежит результат. Но с помощью циркуля и линейки решается не только задача деления отрезка пополам, но и целый ряд геометрических построений. С помощью арифметических операций также строятся алгоритмы решения самых различных задач.

Набор средств и понятий, позволяющих строить не один алгоритм, а множество алгоритмов, решающих различные задачи, будем называть *алгоритмической системой*. Алгоритмических систем может быть много и каждая из них определяется:

1) множеством входных объектов, подлежащих обработке алгоритмами данной системы;

2) свойствами исполнителя алгоритмов, т. е. набором тех действий, которые может выполнять исполнитель;

3) множеством, к которому могут принадлежать результаты выполнения алгоритмов данной системы;

4) языком, на котором формулируются алгоритмы, адресованные исполнителю.

Заметим, что множество входных объектов (исходных данных) и множество результатов часто совпадают. Например, в алгоритмах вычислений арифметических выражений и исходные данные, и результаты — числа. Но в ряде случаев исходные данные и результаты могут оказаться совершенно разной природы. Например, во многих задачах управления в качестве исходных данных алгоритмов, решающих задачи планирования, выступают числа, выражающие производительность, мощность предприятий, запасы ресурсов. Результатом же работы алгоритма может оказаться информация, приводящая к решению построить новый завод.

Другой пример. Исходными данными для алгоритма поиска нужной книги в книгохранилище служат фамилия автора, название книги, год и место ее издания.

Ответом служат числа, указывающие номер стеллажа и номер полки, где следует искать книгу.

Что касается действий, которые может выполнять исполнитель, то они также могут резко отличаться в различных алгоритмических системах. Например, в задачах геометрических построений предполагается, что исполнитель владеет циркулем и линейкой — может проводить прямые линии, выбирать произвольную точку, соединять точки, чертить окружности, отыскивать точки пересечения. В алгоритмах управления роботом-манипулятором в зависимости от конструкции робота предполагаются известными те элементарные движения, которые он может выполнять, например поворот механической руки, движение вперед-назад, захват детали и т. д.

Язык алгоритмической системы тесно связан с исполнителем и не должен включать в свой состав указаний на недопустимые или невозможные для исполнителя действия, а также обращения к входным объектам, если они не принадлежат алгоритмической системе (это же касается и результатов). Он должен быть точно понятим исполнителем.

Если исполнитель алгоритма — человек, владеющий русским языком, то в качестве языка для формулировки алгоритмов вполне можно использовать русский язык.

В алгоритмических системах, предназначенных для построения алгоритмов обработки данных, т. е. обработки символьных последовательностей любого происхождения, в качестве множества исходных объектов рассматриваются наборы символов конечной длины. А как мы уже знаем, наборами символов можно обозначать числа, слова

текста, предложения, т. е. представлять информацию любого содержания.

Результат обработки данных также представляет собой наборы символов. Тем самым в этих алгоритмических системах множество исходных данных и множество результатов совпадают. А вот наборы действий, которые может выполнять исполнитель в различных алгоритмических системах для обработки данных, весьма сильно различаются.

Если исполнитель — человек, то в качестве одного действия ему можно, например, предложить выбрать максимальное число из трех заданных чисел.

В то же время вычислительная машина в одно действие не может решать такую простую задачу, и такое действие для нее превращается в несколько шагов алгоритма. Набор ее операций весьма ограничен, однако, комбинируя их в нужной последовательности, можно построить весьма сложные алгоритмы решения множества различных задач.

Соответственно язык, на котором задаются шаги заданий, адресованных компьютеру, значительно беднее, чем язык, адресованный человеку, так как набор действий, которые может выполнять компьютер, сравнительно невелик. В то же время такой язык более точен, его предложения не допускают различных толкований и алгоритм, написанный на таком языке, представляет собой последовательность команд, адресованных различным устройствам ЭВМ, совсем не похожую на привычные нам фразы естественного языка. По своей сути язык ЭВМ формален, а то что он все-таки назван языком, имеет под собой довольно глубокое основание. Несмотря на несхожесть его с обычным языком, он имеет свой алфавит, свою грамматику.

Алгоритм, адресованный ЭВМ как исполнителю, представляет собой некий текст, т. е. некоторую последовательность символов. В связи с этим отметим одно очень важное обстоятельство. Если мы имеем дело с алгоритмической системой, предназначенной для составления алгоритмов обработки данных, т. е. любых последовательностей символов, то открывается следующая принципиально важная возможность.

Как мы уже отметили, четвертой составной частью алгоритмической системы является язык формулировки алгоритмов. А текст, в том числе текст алгоритма, на любом языке есть также символьная последовательность, которую можно преобразовывать в той же алгоритмической системе. Следовательно, открывается возможность составлять алгоритмы, которые в свою очередь будут преобразовывать алгоритмы, обрабатывая тексты, реализующие эти алгоритмы.

При решении задач с помощью ЭВМ этим широко пользуются, автоматически преобразуя алгоритмы из одной формы в другую или автоматически меняя алгоритм в ходе вычислений. Это создает ту

удивительную логическую гибкость, которая и превратила ЭВМ в принципиально новый инструмент обработки данных. Недаром говорят об искусственном интеллекте, об интеллектуальных системах, построенных на основе ЭВМ, и недаром пытаются с помощью ЭВМ моделировать работу мозга человека.

Итак, в алгоритмической системе можно строить самые различные алгоритмы. Но не всегда конкретный алгоритм, построенный в выбранной алгоритмической системе, будет пригоден для работы со всеми ее входными объектами. Например, в алгоритмической системе, предназначенной для обработки числовой информации, в качестве исходных данных определены все числа. В тех алгоритмах, которые включают в себя операцию деления, в качестве исходных данных нельзя использовать нуль, хотя он и принадлежит множеству исходных данных алгоритмической системы. Тем самым для конкретного алгоритма не все исходные данные оказываются допустимыми. В этих случаях говорят, что данный конкретный алгоритм неприменим к таким исходным данным.

Например, в алгоритмической системе геометрических построений с помощью циркуля и линейки можно рассмотреть задачу построения треугольника по трем заданным отрезкам. Исходными данными в этой системе является множество троек всевозможных отрезков. Однако не любая такая тройка отрезков будет допустимой для алгоритма построения треугольника. Хорошо известно, что треугольник построить нельзя, если длина какого-либо отрезка окажется больше суммы длин двух других отрезков. Поэтому можно сказать, что алгоритм построения к этим исходным данным неприменим, а сами исходные данные являются для него недопустимыми.

Мы уже говорили, что при выполнении алгоритма результат должен быть получен за конечное число шагов. Если же окажется, что при применении алгоритма к каким-либо исходным данным алгоритмический процесс продолжается бесконечно, то и в этом случае говорят, что алгоритм к этим исходным данным неприменим, а сами исходные данные являются для него недопустимыми.

Рассмотрим, например, алгоритм деления двух чисел. И пусть наша цель состоит в том, чтобы при делении двух чисел получить при этом абсолютно точный результат. Возникает вопрос: для всех ли чисел алгоритм деления уголко дает точный результат за конечное число шагов?

Если мы возьмем в качестве делимого и делителя числа 6,3 и 3, то алгоритм деления даст

$$\begin{array}{r|l} 6,3 & 3 \\ \hline 6 & 2,1 \\ \hline 3 & \\ \hline 3 & \\ \hline 0 & \end{array}$$



т. е. искомый точный результат равен 2,1 и получен за конечное число шагов алгоритмического процесса.

Если же мы возьмем в качестве делимого и делителя соответственно числа 2,2 и 3, то получим следующий процесс:

$$\begin{array}{r|l} 2,2 & 3 \\ -2,1 & 0,733... \\ \hline 10 & \\ -9 & \\ \hline 10 & \end{array}$$

Очевидно, что данный процесс никогда не закончится (если не считать, что, найдя период дроби, мы получим результат). Таким образом, для исходных данных 2,2 и 3 невозможно получить искомый результат за конечное число шагов. Произвольный же обрыв процесса дает только приближенное значение, которое всегда будет отличаться от искомого точного результата. Тем самым выбранные исходные данные 2,2 и 3 являются недопустимыми для алгоритма деления уголко. Удивленный читатель может, естественно, спросить, что же значит — нельзя делить уголко 2,2 и 3? Конечно же, можно, но надо иметь при этом в виду, что придется рано или поздно прекратить процесс деления и ограничиться получением приближенного значения результата. Точный же результат получить не удастся.

Отметим как важный факт, что такие задачи, в которых точное решение нельзя получить за конечное число шагов, встречаются очень часто. В таких случаях пытаются найти алгоритм, позволяющий добиться приближенного значения результата с нужной степенью точности за конечное число шагов. Это количество шагов, как правило, зависит как от исходных данных задачи, так и от того, с какой точностью необходимо получить решение — чем выше точность, тем больше вычислений придется проделать. Так было и в нашем примере с делением — чем больше нужно получить знаков в результате, тем длительнее процесс вычислений.

Пусть теперь есть задача, для которой мы нашли какой-либо алгоритм ее решения. Мы знаем, что он может оказаться не единственным и, возможно, далеко не лучшим. Что значит улучшить алгоритм? Это понятие не вполне определенное. Интуитивно ясно, что желательно сократить количество действий для достижения результата и, по возможности, упростить сами эти действия.

Однако при этом алгоритм должен оставаться, как говорят, *эквивалентным* исходному. Сформулируем более четко, что следует понимать под эквивалентностью алгоритмов. Будем называть два алгоритма эквивалентными, если:

1) множество допустимых исходных данных одного из них является множеством допустимых исходных данных и для другого; из применимости одного алгоритма к каким-либо исходным

данным следует применимость и другого алгоритма к этим исходным данным;

2) применение этих алгоритмов к одним и тем же исходным данным дает одинаковые результаты.

Эквивалентны, например, приведенные ранее алгоритмы подсчета количества зрителей на трибунах стадиона по секторам и по рядам трибун.

Хорошо известны преобразования, приводящие квадратный трехчлен  $ax^2 + bx + c$  ( $a \neq 0$ ) к виду, удобному для его исследования:

$$ax^2 + bx + c = a \left( x^2 + 2 \frac{b}{2a} x + \frac{b^2}{4a^2} - \frac{b^2}{4a^2} \right) + c = a \left( x + \frac{b}{2a} \right)^2 - \frac{b^2 - 4ac}{4a}.$$

Исходная формула и формула, полученная в результате преобразований, задают эквивалентные алгоритмы вычисления значения квадратного трехчлена. Для любых  $a, b, c$  (кроме  $a=0$ ) эти трехчлены определены на всей числовой оси (т. е. области допустимых исходных данных у них совпадают) и задают для одинаковых значений  $x$  одинаковые значения функций.

Эквивалентными являются и алгоритмы для вычисления значения многочлена в некоторой точке  $x$ , заданные следующими формулами:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \dots + a_1 x + a_0, \quad (1)$$

$$P_n(x) = (\dots((a_n x + a_{n-1}) x + a_{n-2}) x + a_{n-3}) x + \dots + a_1) x + a_0 \quad (2)$$

(запись (2) носит название схемы Горнера). Для одного и того же набора коэффициентов обе формулы дают для фиксированного значения  $x$  одинаковые результаты. Чтобы убедиться в этом, достаточно раскрыть скобки в формуле (2).

Формулы (1) и (2) определяют алгоритмы для вычисления значения многочлена  $P_n(x)$ . В формуле (1) сначала вычисляется  $x^2 = x \cdot x$ , затем  $x^3 = x^2 \cdot x$ ,  $x^4 = x^3 \cdot x$ , ... и т. д. Полученные результаты домножаются на соответствующие коэффициенты  $a_i$ :  $a_1 \cdot x$ ,  $a_2 \cdot x^2$ ,  $a_3 \cdot x^3$ , ... Затем все это складывается. Например, чтобы вычислить с помощью формулы (1) значение многочлена четвертой степени  $P_4(x)$  в точке  $x$ , необходимо проделать следующие вычисления: вычислить  $x^2$ ,  $x^3$ ,  $x^4$  (на это уйдет три операции умножения), затем получить значения  $a_1 x$ ,  $a_2 x^2$ ,  $a_3 x^3$  и  $a_4 x^4$  (это еще четыре операции умножения) и, наконец, посчитать сумму

$$a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

(еще четыре арифметические операции). На все это потребовалось 11 арифметических операций.

Если же мы вычислим значение многочлена  $P_4(x)$  в точке  $x$  по формуле (2), т. е.

$$P_4(x) = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0,$$

то нам придется вычислить подряд следующие значения:

$$\begin{aligned} a_4x, & \quad a_4x+a_3, \quad (a_4x+a_3)x, \quad (a_4x+a_3)x+a_2, \\ ((a_4x+a_3)x+a_2)x, & \quad ((a_4x+a_3)x+a_2)x+a_1, \\ (((a_4x+a_3)x+a_2)x+a_1)x, & \\ (((a_4x+a_3)x+a_2)x+a_1)x+a_0. & \end{aligned}$$

Нетрудно посчитать, что для вычисления значения  $P_4(x)$  нам пришлось выполнить 8 арифметических операций.

Итак, алгоритмы вычисления значения многочлена, определяемые формулами (1) и (2), эквивалентны. Но алгоритм, задаваемый формулой (2), требует меньше арифметических операций. В дальнейшем мы убедимся, что он и значительно удобнее при реализации на ЭВМ.

### § 1.3. Понятие о математических моделях

Мощным инструментом познания мира является математическое моделирование. «Математическая модель — приближенное описание какого-либо класса явлений внешнего мира, выраженное с помощью математической символики» (академик А. Н. Тихонов). Всякое явление природы бесконечно в своей сложности. Чтобы описать явление, необходимо выявить самые существенные его свойства, закономерности, внутренние связи, роль отдельных характеристик явления. Выделив наиболее важные факторы, влияющие на происходящие процессы, можно пренебречь менее существенными. Тем самым мы несколько упрощаем и огрубляем модель явления.

Так, например, рассмотрим движение камня, падающего с некоторой высоты на землю. Естественно, при падении ему приходится преодолевать сопротивление воздуха. Можно ли считать, что скорость падения камня определяется лишь высотой, с которой он падает, и силой притяжения земли? Иначе говоря, можно ли пренебречь силой сопротивления воздуха? Оказывается, это зависит от того, с какой высоты падает камень. Если высота мала и он не успел разогнаться, то скорость его мала и сопротивление воздуха незначительно сказывается на скорости падения. Если же камень летит с большой высоты, то сила сопротивления воздуха, которая, как известно, пропорциональна квадрату скорости падения камня, становится весьма существенной и сильно сказывается на скорости его падения.

Можно поставить и такой эксперимент: из некоторого замкнутого сосуда выкачать воздух и внутри этого сосуда с одинаковой высоты одновременно отпустить в свободное падение два тела — тяжелый камень и легкую пушинку. Независимо от высоты, с которой падают эти тела, приземлятся они одновременно. Пройденный ими путь и время падения будут связаны соотношением

$$S = gt^2/2,$$

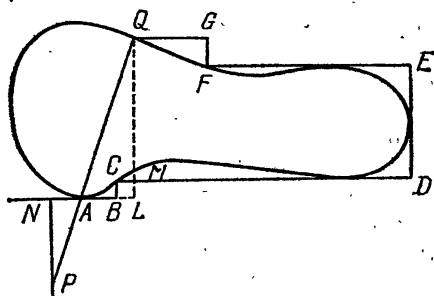
где  $g \approx 9,8 \text{ м/с}^2$  — ускорение свободного падения. Это соотношение и есть математическая модель, описывающая свободное падение тел.

Причем модель тем более точно описывает физическое явление, чем меньше сопротивление воздуха, т. е. либо чем более разрежен воздух, либо чем меньше высота, с которой начинает падать тело. Таким образом, возникает вопрос о границах применимости модели.

Учет важнейших факторов, влияющих на изучаемое явление, и взаимосвязей этих факторов производится с помощью математической символики, как правило, в виде уравнений и неравенств. Получившиеся математические соотношения в совокупности с некоторыми известными исходными данными и образуют математическую модель явления. Исследуя соответствующую задачу, пытаются проникнуть в сущность изучаемого явления, прогнозировать явление, воздействовать на него, управлять им.

Конкретное явление природы, науки, техники переводится на язык математики, и дальше уже исследуется сугубо математическая проблема. Так было в древности, и тем более широко этот метод используется теперь.

В VI в. до н. э. Эвпалин построил на острове Самос водопровод. Ему пришлось прорыть туннель длиной 1 км, а шириной и высотой 2 м сквозь гору Кастро (туннель сохранился до сих пор). Нужно было соединить туннелем точки A и Q



(рис. 1.2). Поступил Эвпалин следующим образом. Он соединил точки A и Q «прямоугольной» ломаной линией, окаймляющей препятствия, как показано на рис. 1.2. Такой же рисунок был, вероятно, и у Эвпалина. На этом рисунке он продлил отрезок AB до пересечения в точке L с перпендикуляром, опущенным из точки Q на продолжение AB, образовав треугольник AQL. Затем на местности измерил длины отрезков AB, CD, QG, FE, а также отрезков GF, ED, CB. Вычтя из суммы длин отрезков AB и CD сумму длин отрезков QG и FE, он получил длину отрезка AL, а сложив длины отрезков GF, ED и CB, получил длину отрезка QL. Далее на продолжении AB (влево) взял произвольную точку N и построил в этой точке перпендикуляр к AB. Отметил на этом перпендикуляре точку P такую, что

$$\frac{|NP|}{|NA|} = \frac{|QL|}{|AL|}.$$

Теперь осталось только прорыть туннель в направлении, указанном отрезком PA.

В своем рассуждении Эвпалли исходил из факта подобия треугольников  $AQL$  и  $APN$ . Разумеется, аналогичным образом легко найти направление туннеля и в точке  $Q$ . Древний математик пренебрег некоторыми реальными деталями. Он считал, что ломаный путь проходит по идеально ровной долине, перегороженной горой. В действительности, наверное, это было не так, но, по-видимому, уклонами можно было пренебречь и с практически приемлемой точностью равнину, окружающую гору, можно было считать плоскостью. Любопытно, что когда туннель был прорыт, то отклонение его выхода от точки  $Q$  составило менее 10 м, т. е. погрешность не превысила 1 %.

Еще до Архимеда было хорошо известно свойство параболы: поток лучей, параллельных оси параболы, после отражения от параболы пересекается в ее фокусе. Архимед нашел этому свойству практическое применение и создал параболические зеркала, с помощью которых был сожжен римский флот (рис. 1.3). Это еще один пример использования математической модели для решения практической задачи.

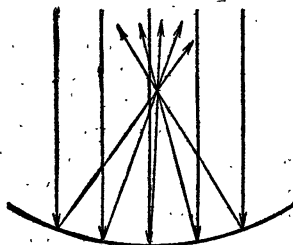


Рис. 1.3

Математическая модель какого-либо явления не обязательно есть что-то консервативное, застывшее. Она может развиваться, совершенствоваться.

Характерным примером такого рода является модель Солнечной системы. Еще во II в. н. э. древнегреческий ученый Птолемей разработал геоцентрическую модель, исходящую из того, что планеты и Солнце вращаются вокруг неподвижной Земли. Пользуясь этой теорией, уже можно было составить алгоритм вычисления положения небесных тел. Теория была очень сложна и все усложнялась, чтобы быть лучше согласованной с результатами наблюдений. Лишь в 1543 г. Коперник сделал крупный шаг вперед, построив гелиоцентрическую модель в предположении, что планеты движутся вокруг Солнца по окружностям (эпициклам). Эта модель позволила вычислять более точно движения планет и объяснять многие непонятные ранее астрономические явления. Однако и Коперник и Птолемей были в плену механики Аристотеля, признававшей только простое движение, т. е. по прямой и по окружности. Это сильно сковывало теорию Коперника.

Следующий шаг в развитии модели Солнечной системы связан с именем Кеплера (начало XVII в.), сформулировавшего три закона движения планет. И, наконец, Ньютон (вторая половина XVII в.) предложил динамическую модель Солнечной системы, основанную на законе всемирного тяготения. Модель стала настолько совершенной, что в 1846 г. Леверье вычислил местоположение новой планеты, и она была открыта точно в том самом месте, которое указал астроном. Он исходил из того, что в движении планеты Уран были замечены

некоторые противоречия, которые бы снимались, если бы в определенном месте была планета, которую он назвал Нептуном. Ученый сумел вычислить массу планеты и законы ее движения. Позднее, в 1930 г., аналогичным образом была открыта планета Плутон. Эти планеты, как говорят, были открыты на кончике пера. Таким образом, динамическая модель Солнечной системы позволяет строить алгоритмы, пользуясь которыми можно с большой точностью определять местонахождение планет, их массы, законы их движения.

Процесс построения математической модели обычно состоит из нескольких этапов. О первом этапе мы, по существу, уже говорили. Он состоит в том, что выделяются основные закономерности, характеризующие моделируемое явление. Эти закономерности зачастую носят гипотетический характер. Так, в примере с моделью Солнечной системы сначала исходили из предположения, что Солнце вращается вокруг Земли, затем из того, что Земля вращается вокруг Солнца; и т. д. Затем закономерности облакаются в математическую форму. На этом этапе возникает целый ряд сложностей. С одной стороны, хотелось бы выявить и формализовать все факторы, влияющие на изучаемое явление. С другой стороны, запись их в виде уравнений и неравенств обычно сопряжена с большими сложностями. Дело в том, что информацию для составления математических соотношений, как правило, приходится черпать из наблюдений. Результаты же наблюдений всегда сопровождаются некоторыми погрешностями, которые тем самым привносятся и в математические соотношения. Как следствие может оказаться, что решение одних уравнений или неравенств, характеризующих данное явление, не удовлетворяет другим соотношениям той же самой модели. Конечно же, такая модель должна быть изменена, т. е. на этом этапе составления математической модели главное — выявить все основные закономерности явления, максимально точно замерить все его параметры, записать все это в математической форме. При этом необходимо следить за тем, чтобы получившиеся соотношения были совместными, т. е. чтобы существовало решение задачи.

Следующий этап состоит в решении поставленной математической задачи. Это уже проблема чисто математическая. Решив задачу, необходимо проанализировать результаты ее решения. Ведь, как мы говорили, модель является гипотетической, основанной на некоторых принятых предположениях. Поэтому нужно убедиться в том, что ее выводы согласуются с результатами наблюдений, т. е. проверить истинность теории практикой. Если окажется, что результаты вычислений не соответствуют наблюдениям, то модель придется забраковать. Так, если, пользуясь какой-то моделью Солнечной системы, мы попытаемся определить траекторию планеты и эта траектория окажется отличной от полученной в результате наблюдений, то такая модель требует изменений. Придется воспользоваться какими-то

другими гипотезами, написать соответствующие математические соотношения, несколько иначе описывающие исследуемое явление, т. е. построить новую математическую модель. Может случиться, что и на этот раз результаты окажутся неудовлетворительными. Тогда опять надо поправить модель. И так далее. Разумеется, все это делается в предположении, что сама математическая задача, возникающая для каждого варианта модели, решается достаточно точно и если возникают расхождения с ожидаемыми результатами, то только из-за недостаточно удачно выбранных гипотез, положенных в основу модели.

В действительности, и решение математической задачи зачастую вызывает значительные трудности. Поэтому при построении модели необходимо учитывать, к какой математической задаче она сведется, — имеет ли эта задача решение, легко ли получить это решение. Так, уже упоминавшаяся проблема прогноза погоды требует учета огромного количества факторов, ее математическая модель сводится к сложным уравнениям, решить которые в короткий срок удастся только с применением электронных вычислительных машин.

Весьма сложными как для построения математических моделей, так и для решения получающихся при этом задач являются проблемы создания сверхмощных ускорителей и запуска космических аппаратов, проектирования новых машин и устройств, в частности новых ЭВМ, и управления большими производственными объединениями, задачи кристаллографии и структурной лингвистики, задачи распределения ресурсов в экономике и планирования народного хозяйства, задачи геодезии и картографии. О компьютерной томографии — принципиально новом виде рентгенологического исследования, при котором получаемую информацию обрабатывает ЭВМ, — говорят как о революции в медицинской диагностике. Перечень таких проблем исключительно важных и столь же исключительно сложных как с точки зрения построения математических моделей, так и с точки зрения решения соответствующих задач можно продолжать неограниченно. Причем найти удовлетворительное решение этих трудоемких задач можно только с помощью ЭВМ. При этом не следует забывать, что все, что делает машина, она делает не сама по себе, а только в соответствии с указаниями, которые ей дает человек, т. е. в соответствии с алгоритмом, заложенным в машину.

Таким образом, человек составляет алгоритм решения задачи, записывает его в форме, понятной данной ЭВМ, т. е. в виде так называемой программы, и далее уже машина действует в соответствии с этой программой. Поэтому, когда мы говорим, что машина играет в шахматы, это означает, что построена математическая модель шахматной игры, т. е. игра переведена на язык математической символики, в машину заложены многочисленные дебютные схемы, известные варианты и т. д. Построен алгоритм, который анализирует ситуацию — текущее положение фигур на доске, и выбирает наилучший

ход. Кстати, в первом чемпионате мира по шахматам среди ЭВМ победила советская машина, а точнее говоря, программа «Каисса».

Электронно-вычислительная техника позволила развить и обогатить саму математику. Появилась возможность решать задачи, которые ранее не решались из-за их невероятной трудоемкости. Получили новое развитие вычислительная математика, математическая физика. Появились новые разделы математики — дискретная математика, теория игр и т. д.

Для решения старых классических задач — таких, например, как решение систем линейных алгебраических уравнений, были созданы и продолжают создаваться новые методы решения и соответствующие алгоритмы. Тем более это необходимо для решения вновь возникающих проблем. Создаются, в частности, такие методы, которые мало чувствительны к погрешностям исходных данных, что исключительно важно при решении задач практических. Такие алгоритмы стараются делать оптимальными, т. е. чтобы ЭВМ, работая по этим алгоритмам, получала как можно более точный результат, затрачивая минимум времени и машинных ресурсов. Естественно, такие уже созданные оптимальные алгоритмы должны стать всеобщим достоянием.

Поэтому существует фонд алгоритмов и программ, в который заносятся разработанные для решения задач науки и техники алгоритмы и программы, удовлетворяющие самым строгим критериям. Они оформляются по определенному стандарту, чтобы каждый мог ими пользоваться при решении своих конкретных задач. Так, если нам нужно решить какую-то свою вполне определенную систему линейных алгебраических уравнений, то нам нет необходимости составлять алгоритм решения, мы можем выбрать из многочисленных уже существующих алгоритмов тот, который нам больше подходит, и применить его к нашим исходным данным.

Нередко алгоритм решения какой-либо большой задачи составляется из последовательности уже ранее известных алгоритмов, это ускоряет процесс составления общего алгоритма, а главное значительно повышает его качество и надежность. Ведь каждый из используемых алгоритмов создавался специалистами именно в этой области математики и проходил самую строгую проверку.

## § 1.4. Алгоритмизация

Под *алгоритмизацией* понимают процесс разработки алгоритма решения какой-либо задачи. Переход от содержательной постановки задачи к разработке алгоритма ее решения иногда вызывает большие трудности. Так, например, простейшая задача — составить алгоритм отыскания максимального числа среди заданной последовательности  $N$  чисел — уже вызывает затруднение скорее всего психологического характера. «Что здесь придумывать — посмотрел и выбрал». Ну, хо-



рошо, если нужно выбирать из трех чисел. А если из тысячи? Тогда уже вовсе не просто «посмотреть и выбрать». Нужна четкая последовательность действий — алгоритм. Его легче проиллюстрировать на примере.

Пусть выстроена шеренга из тысячи человек. Как выбрать среди них самого высокого? Сначала пусть выйдет вперед первый в шеренге. Сравним его по росту со вторым. Тот из них, кто выше ростом, выйдет вперед, а тот, кто меньше, встанет в строй. Если они одинакового роста, то оставим их на своих местах. Теперь стоящего впереди сравним с третьим в шеренге. Тот, кто выше из них, выйдет вперед, а меньший вернется в строй. Теперь уже перед строем будет стоять самый высокий из первых трех в шеренге. Его сравним с четвертым. Тот, кто выше из них, выйдет вперед, — он теперь самый высокий из первых четырех в шеренге. И так далее, до тех пор, пока не будут просмотрены все люди в шеренге. По окончании просмотра перед шеренгой окажется самый высокий человек.

Естественно, ту же схему алгоритма можно применить и для отыскания максимального из  $N$  чисел. Здесь сначала в качестве максимального числа берется первое по порядку в последовательности чисел. Оно сравнивается со вторым, и большее из них принимается за максимальное. Если числа одинаковые, то за максимальное принимается, для определенности, первое. Выбранное число сравнивается с третьим в последовательности. Теперь большее из них становится максимальным среди трех чисел. Это число сравнивается с четвертым в последовательности, и опять-таки большее из них становится максимальным, но теперь уже из четырех чисел последовательности. И так до тех пор, пока не переберем все члены последовательности, каждый раз сравнивая максимальные из предыдущих чисел с очередным членом последовательности. По окончании перебора получим максимальное число в заданной последовательности чисел. Этот алгоритм приведет к правильному результату независимо от количества членов последовательности.

Разработчик алгоритма всегда должен четко представлять себе ту алгоритмическую систему, в рамках которой составляется алгоритм. Если речь идет о составлении алгоритма для ЭВМ, нужно проанализировать, справится ли данная ЭВМ с работой, достаточна ли будет точность вычислений, выполняемых конкретной ЭВМ, достаточен ли будет объем ее запоминающих устройств, хватит ли ее быстродействия для получения ответа в приемлемый срок. Иными словами, при разработке алгоритма нужно хорошо представлять возможности исполнителя алгоритма и тщательно их учитывать.

В предыдущем параграфе мы говорили о методах построения математических моделей. Процесс алгоритмизации по своей методике очень близок к процессу построения математических моделей. Мало того, при построении собственно формального математического описа-

ния явления и задач, которые нужно решать в связи с анализом такого явления, на современном этапе необходимо прямо учитывать возможность построения такого алгоритма решения, который могла бы выполнить имеющаяся в распоряжении исследователя ЭВМ.

В этом смысле математическое моделирование и процесс алгоритмизации теснейшим и неразрывным образом связаны между собой.

Процесс алгоритмизации мы рассмотрим на некоторых примерах. Пусть перед вами поставлена задача разработать алгоритм отыскания номера телефона по фамилии абонента. Метод решения очень прост. В память ЭВМ, как в записную книжку, заносятся фамилии абонентов в алфавитном порядке и рядом с каждой фамилией ставится соответствующий номер телефона. В качестве входных данных алгоритма используется фамилия, набираемая на алфавитно-цифровой клавиатуре. Алгоритм состоит в том, что производится сравнение заданной фамилии на совпадение с фамилией, записанной в памяти машины, и когда такое совпадение произойдет, то на печать или экран ЭВМ следует вывести записанный рядом номер телефона. Однако в ЭВМ нет отдельной операции, позволяющей сравнивать между собой слова произвольной длины, а есть только операции побуквенного сравнения, и, следовательно, нужно продумать, как, используя в шагах алгоритма только операцию побуквенного сравнения, получить ответ, совпадают или не совпадают фамилии, взятые из памяти, с фамилией, набранной на клавиатуре. Разумеется, если память ЭВМ мала, то и компьютерный телефонный справочник не может быть слишком большим. Обычная записная книжка имеет алфавитный указатель. Это обеспечивает быстрый поиск, вы сразу начинаете поиск с нужной буквы, не перелистывая всю записную книгу. При построении алгоритма следует учесть обыденный опыт и не просматривать фамилии одна вслед за другой, начиная с первой, а постараться начинать поиск с нужной буквы или с нужного сочетания букв.

Хотелось бы, чтобы на этом примере задачи поиска, которая на первый взгляд совершенно очевидна по постановке и по методу решения, читатель понял, что разработка алгоритма требует вдумчивости, учета многих факторов, включая технические возможности ЭВМ.

Несмотря на то, что разработка каждого нового алгоритма требует своего собственного подхода, тем не менее есть некоторые общие приемы и этапы этого рода деятельности.

Первый необходимый этап разработки алгоритма может быть охарактеризован как содержательный анализ задачи. На этом этапе следует вникнуть в суть задачи, выяснить, что дано и что требуется получить. Пожалуй, самое важное — это оценить множество исходных данных. Ведь речь, как правило, идет о решении «массовых» задач, о разработке алгоритма, обладающего свойством массовости.

Возьмем пример задачи расчета направления туннеля, который рассматривался ранее. Там шла речь о конкретной местности, которую

пересекал хребет во вполне определенном направлении и заранее известных размеров.

Предположим теперь, что мы хотим разработать такой алгоритм поиска направления туннеля, который был бы пригоден для любой местности с любым расположением хребта и любым его размером. Тогда постановка задачи примет иной вид. А именно, на равнине имеются два пункта  $A$ ,  $Q$  и преграда. Преграда такова, что может загроживать прямой путь из  $A$  в  $Q$ . Требуется построить ломаную линию, состоящую из перпендикулярных отрезков, соединяющую  $A$  и  $Q$  и окаймляющую преграду. Коль скоро такой путь будет построен и отрезки пути будут измерены, мы уже известным способом наметим направление, по которому надо прокладывать туннель.

Содержательный анализ задачи требует ответа на ряд вопросов. Например, может ли быть преграда такой, чтобы она со всех сторон окружала пункт  $A$ ? Если она может быть такой, то задача не имеет решения — ломаную линию в обход препятствия не проведешь. Возможен также случай, что преграда (хребет) обеими концами упирается в побережье моря, и опять в этом случае задача не может быть решена. Наконец, поскольку мы хотим разработать алгоритм, пригодный для всех случаев расположения пунктов  $A$ ,  $Q$  и преграды, надо учесть и такую возможность, что преграда не мешает видеть  $Q$  из  $A$ . Тогда задача решается совсем просто. Содержательный анализ задачи в данном случае говорит нам, что не для любых исходных данных она может быть решена. А в некоторых случаях предложенный первоначально алгоритм не имеет смысла.

Итак, первый этап алгоритмизации, состоящий в анализе задачи, отвечает нам на вопрос: может ли быть задача решена вообще и при каких исходных данных мы можем получить имеющий смысл результат?

Второй этап алгоритмизации состоит в точной постановке задачи или в построении математической модели исходной задачи. Об этом этапе мы подробно говорили в предыдущем параграфе. В рассматриваемом примере мы можем говорить не о равнине, а о плоскости, пункты  $A$  и  $Q$  считать точками на плоскости. Хребет и другие преграды считать линиями, которые нельзя пересекать при построении пути.

Теперь мы уже можем рассматривать чисто геометрическую задачу нахождения направления из  $A$  в  $Q$  с помощью проведения пути, состоящего из взаимно перпендикулярных отрезков, не пересекающих преград. Обратите внимание на следующее: когда мы перешли от равнин и пунктов к геометрической картинке, то, казалось бы, чего проще провести на ней с помощью линейки линию, соединяющую  $A$  и  $Q$ , измерить угол ее наклона, и задача тем самым будет решена. Но тут надо вспомнить исходную задачу: на реальной местности, пересеченной хребтом, линейку между пунктами  $A$  и  $Q$  не проложишь! Поэтому, хотя мы и свели задачу к геометрическим построениям на

плоскости, не всеми возможностями такого построения можно воспользоваться.

Третий этап алгоритмизации можно назвать анализом возможностей исполнителя или более точно — анализом алгоритмической системы, в которой мы будем строить наш алгоритм.

В данном примере с туннелем в качестве исполнителя предполагался человек, который может определять направление движения на север, восток, юг, запад, измерять расстояния, но не может пересекать горные хребты и ходить по морю.

Более подробно приемы алгоритмизации лучше продемонстрировать на другом примере, где в качестве исполнителя выступает не человек, а ЭВМ. Хотя этот пример и более сложен, но он касается одной интересной задачи, связанной с анализом изображений с помощью ЭВМ.

Сегодня с помощью ЭВМ обрабатываются изображения, полученные с искусственных спутников земли, с космических аппаратов, совершающих посадки на другие планеты Солнечной системы. С помощью ЭВМ восстанавливаются старые фотографии, производится раскраска черно-белых кинолент. Появилось целое научно-техническое направление, которое названо *машинной графикой*. Оно сейчас очень бурно развивается и приносит большую реальную пользу. Наш пример будет как раз относиться к машинной графике.

Представим себе следующую задачу. Пусть нам показывают фотографии одинакового формата (например,  $6 \times 9$ ), на которых зафиксировано изображение вертикальной линии. На каждой фотографии эта линия может располагаться в любом месте. Нам требуется определить координаты вертикальной линии, т. е. расстояние до нее от левого края фотографии. Если фотография четкая, то это делается очень просто. Берется линейка и измеряется нужное расстояние в миллиметрах или даже в долях миллиметра, если у нас хороший измерительный инструмент. Если же фотография нечеткая, линия на ней получилась размытая, задача уже несколько усложняется. В этом случае мы скорее всего за результат примем расстояние до середины размытой полосы. В реальных задачах, связанных с анализом фотографий, чаще всего встречается именно вторая ситуация, когда изображение содержит какие-либо дефекты.

Теперь предположим, что нам нужно составить алгоритм обработки разных фотографий вертикальных линий для ЭВМ. При этом мы знаем, что фотографии могут быть и хорошими и плохими, т. е. с дефектами. На первом этапе алгоритмизации, связанном с пониманием существа задачи, нам следует точно выяснить, что значит «плохие» фотографии. Пусть мы это выяснили и пусть дефекты, которые можно ожидать, таковы: нерезкое изображение линии, связанное с плохим наведением фотоаппарата на резкость, возможные царапины на фоне изображения, пятна на фотографии.

Наш алгоритм должен учитывать такие дефекты и, если это возможно, отыскивать на фотографии место расположения вертикали.

Отсюда сразу следует, что не всегда задача имеет решение. Например, если дефект таков, что изображение линии оказалось полностью стертым, то отыскать линию невозможно. Если пятно такое, что все поле фотографии полностью затемнено, то результат тот же. И какой бы мы с вами алгоритм ни придумали, он не сможет решить задачу для таких «плохих» исходных данных. Однако нам надо стремиться к тому, чтобы составленный алгоритм оказывался работоспособным как можно в более широких пределах допустимых дефектов.

Следующее, что нам необходимо понять, — это каким образом фотография окажется в памяти ЭВМ.

В настоящее время разработано много различных устройств для ввода изображений в ЭВМ. Но принципиально эти устройства выполняют одну и ту же работу: они превращают исходную фотографию в мозаику черных и белых пятен, обычно расположенных в узлах прямоугольной сетки, т. е. пятна мозаики следуют строго по горизонталям, образуя строки, и по вертикалям, образуя колонки.

Возьмите увеличительное стекло, посмотрите на газетную фотографию и вы увидите как раз такую мозаику, о которой идет речь. Процесс превращения исходного изображения в мозаику называется квантованием. Чем мельче сетка мозаики, тем точнее мозаичная картина соответствует исходной фотографии. Аппаратура ввода изображений квантует исходное изображение и передает в память ЭВМ координаты всех пятен мозаики и их значение — черное пятно или белое пятно.

Отвлечись на минуту, заметим, что современные устройства ввода изображений передают в ЭВМ значение цвета и освещенности каждого пятна мозаики при вводе цветных и полутоновых изображений.

Мы будем считать, что имеем дело с устройством ввода, которое передает в ЭВМ черно-белое изображение, преобразованное в мозаику черных и белых пятен.

Несколько идеализируя ситуацию, можно вообразить, что в памяти ЭВМ после работы устройства ввода вместо исходной фотографии окажется изображение следующего характера. Представим, что на исходную фотографию мы наложили прозрачную миллиметровую сетку и все ее квадратики, в которые попали достаточно темные части изображения, зачернили, а остальные оставили светлыми. После этого мы передали на ЭВМ, последовательно просматривая квадратик за квадратиком по строкам, информацию следующего содержания. Если нам попался черный квадратик, то мы передали цифру 1, если светлый — цифру 0. Тем самым в памяти ЭВМ окажется своеобразная мозаика, построенная из нулей и единиц, в точности отражающая мозаику на разрисованной прозрачной миллиметровке. Мы будем считать,

что именно в таком виде исходные данные для нашего алгоритма окажутся в ЭВМ.

Зная существо задачи, зная, как и в каком виде исходные данные попадут в ЭВМ, мы можем приступить к построению алгоритма. Сначала нам следует придумать основную идею алгоритма, принцип его работы.

Проанализируем, что нам известно, что неизвестно и что следует вывести в качестве результата и в какой форме. Нам известно, что на исходной фотографии зафиксирована вертикальная линия. Неизвестно, где эта линия расположена. Нам надо восстановить линию. Для этого достаточно определить расстояние от этой линии до левого края фотографии, и тогда задача будет решена. На самом деле, вышеприведенными рассуждениями мы построили математическую модель явления, и в строгой формулировке задача будет выглядеть следующим образом. Дан прямоугольник, на который нанесена прямоугольная сетка. Квадратики сетки помечены либо символом 1, что соответствует черной точке на мозаике, либо символом 0, что соответствует белой точке на мозаике. Требуется найти расстояние от вертикальной линии, изображенной с помощью мозаики, до левого края прямоугольника.

Следующим этапом алгоритмизации является разработка идеи алгоритмического процесса и анализа этой идеи. Первое, что приходит в голову, — это заставить ЭВМ подсчитать число черных точек (единиц) в каждой колонке сетки, найти колонку с максимальным числом черных точек и считать координату этой колонки за место расположения искомого вертикали. Иными словами, идея сводится к тому, чтобы на основе сравнения числа черных точек, встречающихся в колонках, найти расположение искомого вертикали.

ЭВМ хорошо умеет считать и умеет сравнивать, и мы можем быть уверены, что она справится с такой работой, т. е. в принципе выполнение такого алгоритмического процесса можно поручить ЭВМ, коль скоро последовательность из нулей и единиц введена в машину. Очевидно также, что если образ вертикальной линии на мозаике выглядит как одна колонка из черных клеток, то наш алгоритм без особых осложнений отыщет эту колонку.

Мы можем быть уверены также, что наш алгоритм будет работать, если образ линии на мозаике соответствует колонке с максимальным числом единиц (черных пятен), а наличие небольшого числа единиц в других колонках есть отражение разного рода дефектов исходной фотографии. Тем самым мы убеждаемся, что в некоторых ситуациях наш алгоритм будет работать.

Но реально может оказаться так, что образ вертикали превратился на мозаике в несколько соседних колонок, одинаково зачерненных, т. е. в несколько колонок с приблизительно одинаковым числом единичек в них. В этом случае нам надо несколько изменить наш

алгоритмический процесс, считая, что искомая вертикаль расположена в середине «затемненной» полосы, состоящей из нескольких соседних колонок.

Если же дефекты исходной фотографии таковы, что явным образом нельзя выделить колонку или полосу колонок с максимальным числом единиц, то наш алгоритм не сможет дать ответа на интересующий нас вопрос.

По существу, мы тем самым оговорили то множество исходных данных (исходных фотографий), при обработке которых по предположенному алгоритму будет дан ответ.

Мы только что обсудили в самых общих чертах идею алгоритма. Если затем приступить к более детальной его разработке, то снова возникнут вопросы, на которые надо четко сформулировать ответ.

Наш алгоритм на первом этапе дает последовательность целых чисел, показывающих, сколько черных точек находится в первой, во второй, в третьей и т. д. колонках. По этим числам можно построить график частоты встречаемости черных точек по колонкам. У каждой обрабатываемой фотографии свой график. Некоторые графики окажутся с резко выраженным пиком, и в этом случае вполне ясно, что делать. Но могут в принципе встретиться графики с малозаметным горбом. Надо придумать, как должен себя вести алгоритм и в этом случае.

Детальный анализ всех возможных ситуаций, которые могут реально встретиться, представляет собой творческий процесс, требующий не только хорошего знания существа задачи, но и выдумки и изобретательности.

Выше мы рассмотрели пример подхода к алгоритмизации сравнительно простой задачи восстановления вертикальной линии по фотографии с дефектами. Представим себе, что у нас имеется некоторое недостаточно четкое изображение и нам нужно увеличить его контрастность.

Очевидно, что в этом случае надо изобретать совсем иной алгоритм — анализ числа точек по вертикали здесь никак не поможет. Здесь следует искать принципиально иные подходы.

И такие подходы существуют. Поскольку они чрезвычайно интересны и необычны, попробуем рассказать об одном алгоритме, связанном с понятием клеточного автомата.

Представим себе следующий алгоритмический процесс над изображениями, представленными мозаикой черных и белых квадратиков, образуемых квадратной сеткой. У каждого квадратика на сетке есть восемь соседей (см. рис. 1.4).

Будем преобразовывать исходную картинку в новую по следующим правилам. Рассмотрим каждый отдельно взятый квадратик. Он может быть черным или белым. Рассмотрим всех его соседей. Соседние квадратики могут быть также белыми или черными. Если боль-

большинство соседей черные, то на новой картинке мы рассматриваемый квадратик зачерним (если он уже был черный, то так его и оставим). Если большинство соседей оказались белыми, то на новой картинке рассматриваемый квадратик мы сделаем белым вне зависимости от того, каков он был. Если черных и белых соседей одинаковое число (4), то на новой картинке мы оставим цвет рассматриваемого квадрата без изменений, т.е. на новой картинке могут появиться новые черные квадратики (клетки) или исчезнуть черные клетки.

Со вновь полученной картинкой можно проделать то же самое. От шага к шагу будут появляться все новые и новые картинки. В некоторых случаях, правда, дальнейшие шаги могут и не приводить ни к каким изменениям. Легко заметить, что при таком процессе одиночные черные клетки быстро исчезнут, области, густо засеянные черными клетками, зачернятся. Несколько изменив правила возникновения (рождения) черных клеток и их исчезновения (умирания), можно добиться того, чтобы этот процесс приводил к получению контуров областей, можно даже добиться того, чтобы изображение или его отдельные части двигались по полю, как в мультфильме.

Именно этим алгоритмическим процессом широко пользуются при обработке изображений. Строят специальные ЭВМ, предназначенные для эффективной реализации алгоритмических процессов на основе этой идеи. Ведутся серьезные математические исследования в направлении изучения возможностей клеточных автоматов, работающих на рассмотренном выше принципе.

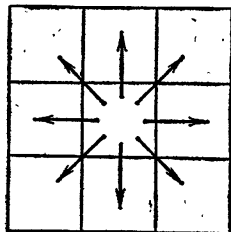


Рис. 1.4



## ГЛАВА 2

### О НЕКОТОРЫХ ВЫЧИСЛИТЕЛЬНЫХ АЛГОРИТМАХ

Построение математической модели какого-либо объекта — лишь первый, хотя и исключительно важный, этап в исследовании этого объекта. Далее приходится иметь дело уже не с самим объектом, а с его математической моделью, которая с той или иной степенью точности его описывает.

Математическую модель уже можно изучать чисто математическими методами, не вдаваясь в физическую природу объекта. Алгоритмы решения возникающих при этом задач иногда бывают просты и их можно выполнить вручную, а иногда столь трудоемки, что без помощи ЭВМ не обойтись.

Результаты решения могут быть получены с той или иной степенью точности, которая зависит, с одной стороны, от того, насколько удачно выбрана математическая модель, а с другой стороны, от точности задания исходных величин, от алгоритма решения задачи, наконец, от тех вычислительных средств, с помощью которых она решается.

Мы хотим привести здесь ряд алгоритмов вычислительного характера. Все они достаточно просты, но, тем не менее, при конкретной реализации некоторых из них возникает необходимость в довольно громоздких вычислениях. В таких случаях помощь вычислительной машины может оказаться весьма полезной.

#### § 2.1. Понятие о приближенных значениях величин

При решении задач вычислительного характера в редких случаях удается получить результат абсолютно точно. Причины этого следующие.

Во-первых, участвующие в операциях числа обычно записываются в виде десятичных дробей. Если исходные числа были иррациональными, то точно представить их в виде конечных десятичных дробей мы, разумеется, не сможем, так как их запись содержит бесконечное число цифр. Поэтому приходится вместо исходных иррациональных чисел оперировать рациональными числами, полученными из

данных иррациональных чисел, если в их записи оставить только первые несколько десятичных знаков.

Так, вместо числа  $\pi$  приходится использовать рациональные числа 3,1 или 3,14, или 3,142, или 3,1416 и т. д. Естественно, что результат действий над такими числами будет содержать некоторую ошибку (погрешность) тем большую, чем меньше десятичных знаков содержат рациональные приближения к исходным иррациональным числам.

Возникают подобные погрешности и тогда, когда исходные данные были рациональными числами. При их записи в виде десятичной дроби может получиться бесконечная периодическая дробь. Так, например, число  $3/7$  представляется в десятичном виде как бесконечная периодическая дробь:

$$3/7 = 0,(428571)$$

(в скобках мы отметили периодически повторяющиеся цифры). Дробь может иметь и конечное число цифр, но их количество может оказаться очень большим.

Имеющиеся в нашем распоряжении вычислительные средства (ЭВМ, калькуляторы) всегда используют ограниченное число цифровых разрядов. Поэтому они не могут учесть все разряды числа, имеющего либо бесконечно много цифр в своей записи, либо конечное, но слишком большое количество цифр. В этих случаях приходится «лишние» цифры отбрасывать, что и ведет к погрешности результата в действиях над такими числами.

Итак, часто бывает, что исходные данные и промежуточные величины мы не можем записать совершенно точно. Это одна из причин неточности ответа.

Во-вторых, одним из самых важных источников погрешности является неточность самих исходных данных. Как правило, исходные данные для задачи получают из какого-либо физического эксперимента или наблюдений. Любой эксперимент связан с измерениями, а измерения всегда производятся с той или иной погрешностью. Например, если нам нужно определить площадь комнаты, мы берем обычную ученическую линейку и измеряем длину и ширину комнаты. Как бы мы ни старались, а на несколько миллиметров, а может быть даже сантиметров, ошибемся, хотя бы из-за того, что сама линейка может быть не абсолютно точной. Естественно, и площадь комнаты как результат умножения длины на ширину получается не совсем точно.

Заметим, что в последнем примере результат будет неточным еще из-за того, что в действительности форма пола комнаты вряд ли представляет собой идеальный прямоугольник — стороны не являются идеальными прямыми, противоположные стороны на самом деле не равны и не параллельны. Поэтому мы вместо реальной задачи решаем другую — идеализированную и более простую. Можно сказать, что мы построили математическую модель, которая приближенно отображает

сущность явления. Естественно, в этом случае результат получится с погрешностью — ведь мы и вычисляем не совсем то, что нужно, и исходные данные в наших вычислениях взяты не точно.

Пусть для величины, точное значение которой есть  $x$ , мы каким-либо образом получили приближенное значение  $x^*$ .

Абсолютная величина разности между точным значением  $x$  и его приближенным значением  $x^*$  называется *абсолютной* погрешностью приближенного числа  $x^*$  и обозначается  $\Delta x^*$ , т. е.

$$|x - x^*| = \Delta x^*.$$

Как правило, точное значение  $x$  нам неизвестно, а следовательно, неизвестна и абсолютная погрешность  $\Delta x^*$ .

Но зато обычно можно определить число, которое эта абсолютная погрешность заведомо не превосходит. Его обычно и принимают за значение абсолютной погрешности измеряемой величины. Оно определяется самим способом нахождения приближенного значения. Так, взвешивая какой-либо предмет на аптекарских весах, мы не сможем определить точность, с которой получен вес предмета, но гарантируем, что ошибка взвешивания не превосходит 0,01 г, т. е. абсолютная погрешность при взвешивании не превышает 0,01 г.

Однако абсолютная погрешность не всегда достаточно полно характеризует погрешность вычислений. В самом деле, пусть ошибка при измерении длины радиоволны равна одному метру. Если при этом измерялась длина волны в диапазоне длинных волн, то точность хорошая, такая же ошибка при измерении длины волны в диапазоне УКВ (ультракоротких волн) слишком велика. Поэтому вводят еще одно важное понятие — относительную погрешность.

*Относительной* погрешностью приближенного значения  $x^*$  называется отношение абсолютной погрешности  $\Delta x^*$  к абсолютному значению приближенной величины:

$$\delta_{x^*} = \Delta x^* / |x^*|.$$

Нетрудно видеть, что если абсолютная погрешность всегда имеет ту же размерность, что и сами величины, то относительная погрешность есть величина безразмерная.

Разумеется, говорить об относительной погрешности можно только в том случае, когда  $x^* \neq 0$ .

В примере с измерением длин радиоволн абсолютная погрешность равна одному метру. Если после измерения получим, что длина волны в диапазоне длинных волн 1000 м, а в диапазоне УКВ 4 м, то в первом случае относительная погрешность не превосходит  $1/1000 = 0,001$ , а во втором —  $1/4 = 0,25$ .

Мы, по-видимому, не будем сильно расстраиваться, если наши часы будут убегать в сутки на 10 с, но вряд ли будем довольны, если они будут убегать на 10 с каждую минуту. В первом случае

относительная погрешность равна

$$\frac{10}{24 \cdot 60 \cdot 60} = \frac{1}{8640} < 0,00012,$$

а во втором —

$$\frac{10}{60} = \frac{1}{6} \approx 0,17.$$

Абсолютная же погрешность и в том и в другом случае одинакова — 10 с.

Выше уже говорилось о том, что на практике часто приходится иметь дело с числами, запись которых требует бесконечно много знаков, и с числами, число знаков у которых конечно, но может быть очень большим. Поэтому такие числа приходится округлять.

Обычно округление производят по следующему правилу (иногда, правда, пользуются и другими правилами).

Пусть какое-то число имеет в своей записи более чем  $k$  цифровых знаков и мы хотим округлить его, оставив ровно  $k$  знаков. Тогда, если  $(k+1)$ -я цифра в записи числа меньше, чем 5, то все цифры, начиная с  $(k+1)$ -й, просто отбрасываются. Например, если в числе  $\pi = 3,14159265358\dots$  мы хотим оставить два знака после запятой, то округленное число будет 3,14; если мы хотим оставить пять знаков после запятой, то получим 3,14159.

Пусть теперь  $(k+1)$ -я цифра больше, чем 5. Тогда тоже отбрасывают цифры, начиная с  $(k+1)$ -й, но в оставшемся числе  $k$ -ю цифру увеличивают на единицу. Так, если в числе  $\pi$  мы хотим оставить шесть цифр после запятой, то получим 3,141593.

Если  $(k+1)$ -я цифра есть 5, а за ней найдется хоть одна отличная от нуля цифра, то поступают как в предыдущем случае, т. е. отбрасывают все цифры, начиная с  $(k+1)$ -й, а  $k$ -ю увеличивают на единицу. В нашем примере, оставив четыре цифры после запятой, получим 3,1416.

Наконец, рассмотрим последний случай, когда  $(k+1)$ -я цифра есть 5, а все последующие за ней цифры — нули. Тогда поступают так: отбрасывают «хвост», начиная с  $(k+1)$ -й цифры, и если цифра четная, то оставляют ее без внимания, если же  $k$ -я цифра нечетная, то увеличивают ее на единицу.

Например, оставив три знака после запятой в числе 5,3865, получим 5,386. Если же округлим число 6,7235, то получим 6,724.

Легко проверить, что абсолютная погрешность числа, полученного после округления (исходное число считаем заданным точно), не превосходит пяти единиц первого отброшенного разряда.

Принято считать, что в приближенном значении величины все цифры верные, если его абсолютная погрешность не превышает половины единицы последнего разряда. При выполнении этого условия можно по записи приближенного значения определить его абсолютную

погрешность. И наоборот, по абсолютной погрешности числа определить число верных знаков в его записи.

Пусть, например, известно, что числа  $x_1=12$ ,  $x_2=1025$ ,  $x_3=3,14$  записаны со всеми верными знаками. Тогда их абсолютные погрешности соответственно не превосходят

$$\Delta x_1=0,5, \quad \Delta x_2=0,5, \quad \Delta x_3=0,005.$$

Если же в результате каких-то вычислений получено число 5,126 и известно, что его абсолютная погрешность  $\Delta x=0,05$ , то в этом случае верными в данном числе являются только первые две цифры: 5, 1.

Производя различные арифметические операции над приближенными числами, мы и ответ получаем приближенно. И погрешность результата зависит как от погрешности исходных данных, так и от тех операций, которые над ними производятся.

Так, например, при сложении и вычитании, чтобы, во-первых, не терять точности вычислений и, во-вторых, не писать слишком много знаков, рекомендуется сохранять в результате столько десятичных цифр, сколько верных знаков имеет наименее точное данное, т. е. то, в котором меньше верных цифр после запятой. Пусть, например, нужно сложить два числа, записанные со всеми верными знаками, 14,2 и 2,428:

$$\begin{array}{r} + 14,2 \\ + 2,428 \\ \hline 16,628 \end{array}$$

В ответе надо оставить лишь 16,6. Поэтому если некоторые исходные величины имеют больше десятичных знаков при сложении и вычитании, чем другие, то их предварительно следует округлить до менее точных, оставляя при вычислении промежуточных результатов лишь одну лишнюю цифру. В окончательном результате эта лишняя цифра должна быть отброшена.

## § 2.2. Решение систем линейных алгебраических уравнений

Известна старинная задача о лошаде и муле: «Лошадь и мул шли с тяжелой поклажей на спине. Лошадь жаловалась на свою непомерно тяжелую ношу. «Чего ты жалуешься? — отвечал ей мул. — Ведь, если я возьму у тебя один мешок, ноша моя станет вдвое тяжелее твоей. А вот если бы ты сняла с моей спины один мешок, твоя поклажа стала бы одинаковой с моей». Скажите же, мудрые математики, сколько мешков несла лошадь и сколько нес мул?»

Если лошадь несла  $x$  мешков, а мул  $y$ , то все решение задачи, разложив ее текст на отдельные части, можно представить следующим образом:

Если я возьму у тебя один мешок,	$x-1$
ноша моя	$y+1$
станет вдвое тяжелее твоей.	$y+1=2(x-1)$
А вот если бы ты сняла с моей спины один мешок,	$y-1$
твоя поклажа	$x+1$
стала бы одинаковой с моей.	$y-1=x+1$

Задача свелась к системе двух линейных алгебраических уравнений с двумя неизвестными:

$$\begin{cases} y+1=2(x-1), \\ y-1=x+1, \end{cases} \quad \text{или} \quad \begin{cases} 2x-y=3, \\ -x+y=2. \end{cases}$$

Решение ее:  $x=5$ ,  $y=7$ . Лошадь несла 5 мешков, а мул — 7.

К системе двух линейных алгебраических уравнений с двумя неизвестными можно свести задачу отыскания уравнения прямой, проходящей на плоскости через две заданные точки:  $(x_1, y_1)$  и  $(x_2, y_2)$  (при условии, что  $x_1 \neq x_2$ ).

Пусть искомое уравнение прямой есть  $y=kx+b$ , где  $k$  и  $b$  — неизвестные величины. Подставляем в это уравнение заданные точки и получаем два уравнения с двумя неизвестными:

$$\begin{cases} kx_1+b=y_1, \\ kx_2+b=y_2, \end{cases}$$

здесь  $x_1, y_1, x_2$  и  $y_2$  — известные величины, а  $k$  и  $b$  нужно найти.

Примеров задач, приводящих к необходимости решения систем линейных алгебраических уравнений, можно приводить сколько угодно. Причем вовсе не обязательно всегда будут получаться два уравнения с двумя неизвестными.

Количество уравнений и количество неизвестных могут быть какими угодно. Они определяются математической моделью задачи. И вовсе не обязательно эти количества должны быть одинаковыми, т. е. число уравнений может быть меньше, больше и равно количе-

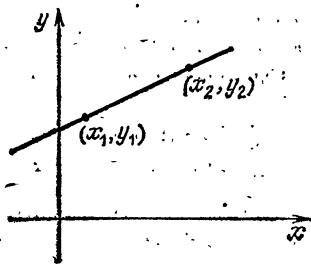


Рис. 2.1

ству неизвестных. Мы здесь будем изучать случаи, когда количество уравнений совпадает с количеством неизвестных. Нужно всегда знать, можно ли решить такую систему, и если можно, то как это сделать.

Рассмотрим сначала систему двух линейных алгебраических уравнений с двумя неизвестными:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1, \\ a_{21}x_1 + a_{22}x_2 = b_2. \end{cases}$$

Неизвестными в системе являются  $x_1$  и  $x_2$ . Заданными — коэффициенты системы  $a_{ij}$  ( $i=1, 2; j=1, 2$ ),  $b_1, b_2$ . Каждый коэффициент  $a_{ij}$  имеет два индекса: 1-й указывает номер уравнения или номер строки системы, а 2-й — номер неизвестного.

Как известно, решением такой системы называется набор чисел  $c_1, c_2$ , который при подстановке их в систему уравнений вместо неизвестных  $x_1, x_2$  соответственно обращает эти уравнения в числовые тождества. Следует помнить, что числа  $c_1, c_2$  представляют одно решение системы уравнений, а не два. Другое дело, что такой набор чисел может быть не единственным.

Вопрос о том, существует ли вообще решение системы уравнений и если существует, то сколько таких решений, для двух уравнений с двумя неизвестными решается просто. Пусть для определенности все  $a_{ij}$  отличны от нуля. Возможны три случая.

1. Если  $a_{11}/a_{21} \neq a_{12}/a_{22}$ , т. е. коэффициенты уравнений непропорциональны, то решение существует и единственно.

2. Если  $a_{11}/a_{21} = a_{12}/a_{22} = k$ , т. е. коэффициенты системы пропорциональны с коэффициентом пропорциональности  $k$ , то возможны два случая:

а)  $b_1/b_2 = k$  (тогда существует бесконечное множество решений);

б)  $b_1/b_2 \neq k$  (в этом случае решения не существуют).

Каждый из рассмотренных случаев имеет простую геометрическую интерпретацию. Дело в том, что каждое из двух уравнений есть уравнение прямой на плоскости. Их легко преобразовать к каноническому виду  $y = kx + b$ :

$$x_1 = -\frac{a_{12}}{a_{11}}x_2 + \frac{b_1}{a_{11}} = k_1x_2 + b_1',$$

$$x_1 = -\frac{a_{22}}{a_{21}}x_2 + \frac{b_2}{a_{21}} = k_2x_2 + b_2',$$

где

$$k_1 = -a_{12}/a_{11}; \quad k_2 = -a_{22}/a_{21}; \quad b_1' = b_1/a_{11}; \quad b_2' = b_2/a_{21},$$

и если  $a_{11}/a_{21} \neq a_{12}/a_{22}$ , то  $-a_{12}/a_{11} \neq -a_{22}/a_{21}$ , т. е.  $k_1 \neq k_2$ .

Итак, в случае 1 прямые имеют разные коэффициенты наклона и пересекутся. Значит, решение существует. Так как прямые могут пересечься лишь в одной точке, то решение единственное (рис. 2.2).

Пусть теперь  $a_{11}/a_{21} = a_{12}/a_{22}$ . Тогда  $-a_{12}/a_{11} = -a_{22}/a_{21}$ , т. е.  $k_1 = k_2$ , и прямые параллельны. Если при этом, как в случае 2а),

$b_1/b_2 = a_{11}/a_{21}$ , то  $b_1/a_{11} = b_2/a_{21}$  и, следовательно,  $b'_1 = b'_2$ . А значит, оба уравнения прямых просто совпадают и прямые имеют бесконечное множество общих точек, а система — бесконечное множество решений (рис. 2.3). Наконец, в случае 2б)  $b_1/b_2 \neq k$ . Тогда  $b'_1 \neq b'_2$  и  $k_1 = k_2$  — две параллельные прямые, которые естественно, не имеют точек пересечения, а, следовательно, система не имеет решений (рис. 2.4).

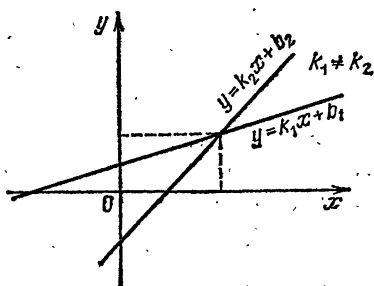


Рис. 2.2

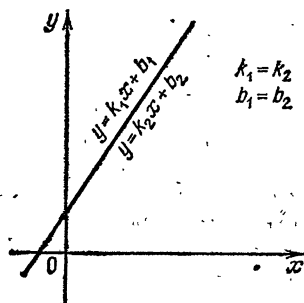


Рис. 2.3

Итак, при решении системы двух линейных алгебраических уравнений с двумя неизвестными возможен один из трех случаев:

- 1) решение существует и единственно,
- 2) решений бесконечное множество,
- 3) решений не существует.

Все сказанное выше можно распространить на системы  $n$  линейных алгебраических уравнений с  $n$  неизвестными для произвольного натурального  $n > 2$ .

Далее мы рассмотрим один из широко распространенных методов решения систем линейных алгебраических уравнений — метод исключения. Он применим к системам  $n$  уравнений с  $n$  неизвестными. Для простоты изложим его на примере системы двух линейных уравнений с двумя неизвестными:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1, \\ a_{21}x_1 + a_{22}x_2 = b_2. \end{cases} \quad (I)$$

Будем считать, по-прежнему, что  $a_{11} \neq 0$ . Если это не так, то можно просто переименовать  $x_1$  в  $x_2$ , а  $x_2$  в  $x_1$ . Кроме того, предположим, что наша система имеет решение и притом единственное, т. е. в соответствии с проведенным нами исследованием это означает, что коэффициенты в левых частях уравнений непропорциональны.

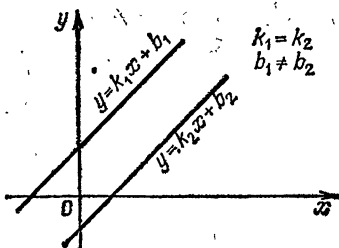


Рис. 2.4



Решать систему (1) будем следующим образом. Прежде всего разделим первое уравнение на коэффициент  $a_{11}$  при неизвестном  $x_1$  (мы предположили, что  $a_{11} \neq 0$ ), оставляя второе уравнение пока без изменений:

$$\begin{cases} x_1 + \frac{a_{12}}{a_{11}} x_2 = \frac{b_1}{a_{11}}, \\ a_{21} x_1 + a_{22} x_2 = b_2. \end{cases} \quad (2)$$

Затем умножим первое уравнение на коэффициент  $a_{21}$  и вычтем это уравнение из второго уравнения системы (2). Получим систему вида

$$\begin{cases} x_1 + \frac{a_{12}}{a_{11}} x_2 = \frac{b_1}{a_{11}}, \\ 0 + \left( a_{22} - a_{21} \frac{a_{12}}{a_{11}} \right) x_2 = b_2 - a_{21} \frac{b_1}{a_{11}}. \end{cases} \quad (3)$$

Второе уравнение системы (3) содержит уже только одно неизвестное  $x_2$ . Из этого уравнения

$$x_2 = \frac{b_2 a_{11} - b_1 a_{21}}{a_{11} a_{22} - a_{12} a_{21}}.$$

Подставив полученное значение  $x_1$  в первое уравнение, получим

$$x_1 = \frac{b_1 a_{22} - b_2 a_{12}}{a_{11} a_{22} - a_{12} a_{21}}.$$

Разумеется, не обязательно запоминать эти формулы для нахождения решения  $x_1$  и  $x_2$ . Можно просто каждый раз при решении конкретных систем уравнений проделывать все так, как здесь описано. Кстати, из наших рассуждений так же с очевидностью вытекает, что в случае, если  $a_{11}/a_{21} \neq a_{12}/a_{22}$ , решение системы уравнений (1) существует и единственно.

Указанный метод решения, как уже говорилось, легко распространить на случай решения систем  $n$  линейных алгебраических уравнений с  $n$  неизвестными:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2, \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n. \end{cases} \quad (4)$$

Идея метода заключается в том, чтобы, оставляя 1-е уравнение системы (4) неизменным, так преобразовать остальные уравнения, чтобы они уже не содержали неизвестного  $x_1$ . Затем, не изменяя 1-го и уже получившегося в результате преобразований при исключении  $x_1$  2-го уравнения, исключить из остальных уравнений 2-е неизвестное  $x_2$ , затем 3-е и т. д., до тех пор, пока получим, что  $n$ -е уравнение содержит лишь одно неизвестное  $x_n$ . Разрешив это уравнение относительно  $x_n$ , можно подставить найденное значение  $x_n$  в предыдущее уравнение, содержащее два неизвестных  $x_n$  и  $x_{n-1}$ . Найти из него  $x_{n-1}$  и перейти к предшествующему уравнению, из которого теперь

можем найти  $x_{n-2}$  и т. д. Наконец, из 1-го уравнения найти  $x_1$ , так как к этому моменту все остальные неизвестные уже найдены.

Мы не будем весь этот процесс описывать в общем виде, а проиллюстрируем его на конкретном примере решения системы четырех линейных уравнений с четырьмя неизвестными:

$$\begin{cases} x_1 + x_2 - 2x_3 + x_4 = 6, \\ 2x_1 + x_2 + x_3 - x_4 = 3, \\ -x_1 - 3x_2 - x_3 + x_4 = -8, \\ x_1 + 2x_2 - x_3 + 2x_4 = 11. \end{cases} \quad (5)$$

Будем постепенно преобразовывать систему. Оставим 1-е уравнение неизменным, 2-е уравнение получим вычитанием из 2-го уравнения системы (5) 1-го уравнения системы (5), умноженного на 2. (Заметим, что если бы коэффициент  $a_{11}$  при  $x_1$  в 1-м уравнении был отличным от единицы, мы бы все первое уравнение поделили на этот коэффициент, и тогда уже коэффициент при  $x_1$  стал бы равным единице.) Таким образом, 2-е уравнение уже не будет содержать неизвестного  $x_1$ . Затем из 3-го уравнения системы (5) вычтем первое уравнение, умноженное на  $-1$ . Значит, и 3-е уравнение новой системы уже не будет содержать  $x_1$ . Наконец, из 4-го уравнения вычтем 1-е, и 4-е уравнение также будет без неизвестного  $x_1$ . Система примет вид:

$$\begin{cases} x_1 + x_2 - 2x_3 + x_4 = 6, \\ -x_2 + 5x_3 - 3x_4 = -9, \\ -2x_2 - 3x_3 + 2x_4 = -2, \\ x_2 + x_3 + x_4 = 5. \end{cases} \quad (6)$$

Теперь, оставляя уже 1-е и 2-е уравнения системы (6) без изменения, избавимся от неизвестного  $x_2$  в 3-м и 4-м уравнениях системы (6). Для этого мы из 3-го уравнения системы (6) вычтем 2-е, умноженное на 2, а из 4-го уравнения вычтем 2-е, умноженное на  $-1$ . Получим

$$\begin{cases} x_1 + x_2 - 2x_3 + x_4 = 6, \\ -x_2 + 5x_3 - 3x_4 = -9, \\ -13x_3 + 8x_4 = 16, \\ 6x_3 - 2x_4 = -4. \end{cases} \quad (7)$$

Наконец, оставив неизменными 1-е, 2-е и 3-е уравнения системы (7), исключим из 4-го уравнения неизвестное  $x_3$ . Для этого поделим 3-е уравнение на  $-13$ , т. е. получим

$$\begin{cases} x_1 + x_2 - 2x_3 + x_4 = 6, \\ -x_2 + 5x_3 - 3x_4 = -9, \\ x_3 - 8/13x_4 = -16/13, \\ 6x_3 - 2x_4 = -4. \end{cases} \quad (8)$$

Теперь умножим 3-е уравнение системы (8) на 6 и вычтем его из 4-го уравнения. Получившееся 4-е уравнение приведем к общему

знаменателю. Система примет вид

$$\begin{cases} x_1 + x_2 - 2x_3 + x_4 = 6, \\ -x_2 + 5x_3 - 3x_4 = -9, \\ x_3 - 8/13x_4 = -16/13, \\ x_4 = 2. \end{cases} \quad (9)$$

Из системы (9), двигаясь от 4-го уравнения к 1-му, получим

$$\begin{aligned} x_4 &= 2; \quad x_3 = -16/13 + 8/13 \cdot 2 = 0; \\ x_2 &= 9 + 5 \cdot 0 - 3 \cdot 2 = 3; \\ x_1 &= 6 - 2 + 2 \cdot 0 - 3 = 1. \end{aligned}$$

Заметим в заключение, что иногда прежде, чем решать систему уравнений, бывает удобно поменять уравнения местами или же переименовать неизвестные так, чтобы удобнее было проделывать все преобразования. Это можно делать и в ходе решения. Теперь читатель может проверить себя — усвоил ли он метод решения системы линейных алгебраических уравнений.

Решить системы уравнений:

$$1) \begin{cases} 2x_1 - x_2 - x_3 = 4, \\ 3x_1 + 4x_2 - 2x_3 = 11, \\ 3x_1 - 2x_2 + 4x_3 = 11. \end{cases}$$

Ответ: 3; 1; 1.

$$2) \begin{cases} 2x_1 + 2x_2 - x_3 - x_4 = 1, \\ 2x_1 + 4x_2 + x_3 - x_4 = 1, \\ -x_1 - x_2 + x_3 + 2x_4 = 2, \\ x_1 + x_2 - 2x_3 - x_4 = 1. \end{cases}$$

Ответ: 0; 1; -1; 2.

$$3) \begin{cases} 3x_1 - x_2 - 2x_3 - x_4 = -3/2, \\ 2x_1 - x_2 + x_3 + x_4 = 2, \\ x_1 - x_2 - 2x_3 + x_4 = 3/2, \\ 4x_1 + 2x_2 - x_3 - 2x_4 = 0. \end{cases}$$

Ответ: 1/2; 1; 0; 2.

## § 2.3. Понятие об интерполировании многочленами

Пусть нам нужно вычислить значение функции  $y = f(x)$  на отрезке  $[a, b]$ , в некоторой точке  $x^*$  этого отрезка, но сама функция нам в виде формулы не задана, а имеется лишь таблица ее значений для некоторого конечного числа отличных друг от друга значений аргумента

$$x_0, x_1, x_2, \dots, x_m, \quad x_i \in [a, b], \quad i = 0, 1, 2, \dots, m.$$

С такой задачей мы сталкиваемся, например, при отыскании логарифма от аргумента, не указанного в таблице логарифмов, а находящегося между двумя соседними значениями табличных аргументов.

Разумеется, если точка  $x^*$  совпадает с какой-либо из указанных точек  $x_i$  ( $i = 0, 1, \dots, m$ ), то для получения необходимого нам

результата достаточно воспользоваться таблицей. Проблема возникает тогда, когда  $x^*$  находится в промежутке между какими-либо из этих точек.

Естественно попытаться найти  $f(x^*)$  следующим образом: подобрать функцию  $y = \varphi(x)$ , заданную формулой, значение которой мы можем вычислить в любой точке отрезка  $[x_0, x_m]$ , такую, что  $\varphi(x_i) = f(x_i)$  ( $i = 0, 1, 2, \dots, m$ ). Тогда можно ожидать, что при удачном выборе  $\varphi(x)$  будет

$$\varphi(x^*) \simeq f(x^*).$$

Есть много причин, по которым  $\varphi(x)$  целесообразно выбирать в виде некоторого многочлена. Одна из них — сравнительная простота вычислений значений многочлена. Можно сузить проблему и искать  $\varphi(x)$  в виде многочлена степени не выше, чем  $m$ . Задача построения такого многочлена называется задачей интерполирования многочленами, а сам многочлен — интерполяционным.

Интерполирование применяют и в том случае, когда функция  $y = f(x)$  имеет представление в виде формулы, но слишком сложное (если нужно получить значения функции для большого количества значений аргумента, то это будет весьма трудоемкой работой). Поэтому вычисляют значения  $f(x)$  в некотором количестве точек и по полученной таблице строят интерполяционный многочлен, которым и заменяют функцию  $y = f(x)$ .

Итак, пусть задана таблица значений функции в точках  $x_0, x_1, \dots, x_m$ . (эти точки называются узлами интерполяции):

$x_0$	$x_1$	$x_2$	$\dots$	$x_m$
$f(x_0)$	$f(x_1)$	$f(x_2)$	$\dots$	$f(x_m)$

Требуется найти многочлен  $P_m(x)$  степени не выше, чем  $m$ , который бы совпал с функцией  $f(x)$  в узлах интерполяции, т. е.  $P_m(x_i) = f(x_i)$  ( $i = 0, 1, \dots, m$ ). Такой многочлен называют интерполяционным многочленом для функции  $f(x)$  по узлам  $x_0, x_1, x_2, \dots, x_m$ . Можно доказать, что такой многочлен существует и единственный. Форма записи его может быть различной. Мы здесь рассмотрим построение интерполяционного многочлена, который называется интерполяционным многочленом Лагранжа.

Для этого вначале построим такой многочлен  $\varphi_0(x)$  степени  $m$ , который принимает значение 1 в узле  $x_0$  и значение 0 в остальных узлах. Таким многочленом будет многочлен

$$\varphi_0(x) = a_0 (x - x_1)(x - x_2) \dots (x - x_m),$$

$$\text{где } a_0 = \frac{1}{(x_0 - x_1)(x_0 - x_2) \dots (x_0 - x_m)}.$$

Действительно,  $\varphi_0(x_i) = 0$  ( $i = 1, 2, \dots, m$ ) для всех  $x_i$ , кроме  $x_0$ , а

$$\varphi_0(x_0) = a_0(x_0 - x_1)(x_0 - x_2) \dots (x_0 - x_m) = 1.$$

Итак,

$$\varphi_0(x) = \frac{(x - x_1)(x - x_2) \dots (x - x_m)}{(x_0 - x_1)(x_0 - x_2) \dots (x_0 - x_m)}.$$

Точно так же строим многочлен  $\varphi_i(x)$ , значение которого равно 1 в узле  $x_i$  и обращается в нуль в остальных узлах:  $\varphi_i(x) = a_i(x - x_0) \times \dots \times (x - x_m)$ . В самом деле,  $\varphi_i(x_j) = 0$  ( $i = 0, 2, 3, \dots, m$ ),  $\varphi_i(x_i) = 1$ , если

$$a_i = \frac{1}{(x_i - x_0)(x_i - x_2) \dots (x_i - x_m)}.$$

Аналогичным образом строим многочлен

$$\varphi_i(x) = a_i(x - x_0) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_m),$$

который обращается в нуль во всех узлах, кроме  $x_i$ . Коэффициент  $a_i$  выражается формулой

$$a_i = \frac{1}{(x_i - x_0)(x_i - x_1) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_m)}.$$

Теперь каждый многочлен  $\varphi_i(x)$  можно записать следующим образом:

$$\varphi_i(x) = \frac{(x - x_0)(x - x_1) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_m)}{(x_i - x_0)(x_i - x_1) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_m)}.$$

Очевидно, что  $\varphi_i(x_i) = 1$  и  $\varphi_i(x_j) = 0$ , если  $i \neq j$  ( $i, j = 0, 1, \dots, m$ ). Если мы рассмотрим произведение  $\varphi_i(x)f(x_i)$  ( $i = 0, 1, \dots, m$ ), то убедимся, что во всех узлах интерполирования, кроме  $x_i$ , оно обращается в нуль (так как  $\varphi_i(x_j) = 0$  при  $i \neq j$ , а в узле  $x_i$  оно равно  $f(x_i)$  (так как  $\varphi_i(x_i) = 1$ ).

Просуммируем теперь эти произведения по всем узлам интерполирования:

$$\varphi(x) = \varphi_0(x)f(x_0) + \varphi_1(x)f(x_1) + \dots + \varphi_m(x)f(x_m).$$

Каждое слагаемое в этой сумме — многочлен степени  $m$ , но если мы эти слагаемые просуммируем и приведем подобные члены, т. е. запишем в виде

$$P_m(x) = a_mx^m + a_{m-1}x^{m-1} + \dots + a_1x + a_0,$$

то может случиться, что некоторые коэффициенты в разложении по степеням  $x$  окажутся равными нулю, в том числе и коэффициенты при старших степенях  $x$ . Иначе говоря, степень многочлена  $P_m(x)$  может оказаться и меньшей, чем  $m$ . Многочлен этот в каждом узле  $x_i$  принимает значение  $f(x_i)$ . Действительно, как мы знаем, слагаемое  $\varphi_i(x)f(x_i) = f(x_i)$ , а остальные слагаемые обращаются в нуль при  $x = x_i$ :

$$\varphi_j(x_i)f(x_i) = 0$$

при  $i \neq j$ .

Многочлен  $\varphi(x)$  называют *интерполяционным многочленом Лагранжа*. Его окончательный вид:

$$P_m(x) = \frac{(x-x_1)(x-x_2)\dots(x-x_m)}{(x_0-x_1)(x_0-x_2)\dots(x_0-x_m)} f(x_0) + \dots \\ \dots + \frac{(x-x_0)(x-x_1)\dots(x-x_{m-1})}{(x_m-x_0)(x_m-x_1)\dots(x_m-x_{m-1})} f(x_m). \quad (10)$$

Построен он по  $(m+1)$ -му узлу интерполяции  $x_0, x_1, \dots, x_m$ , и степень его, как уже говорилось, не выше, чем  $m$ .

В частности, если  $m=1$ , т. е. имеются всего два узла интерполяции, то интерполирование называется *линейным*, так как интерполяционный многочлен является многочленом 1-й степени.

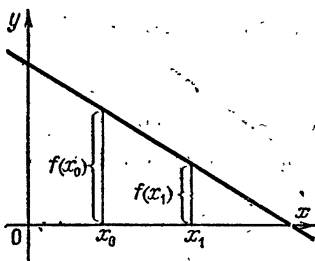


Рис. 2.5

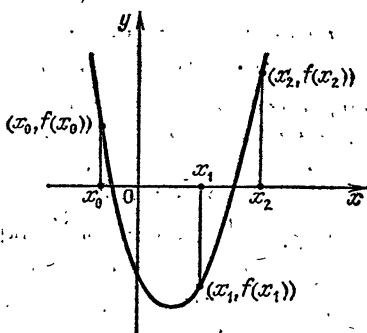


Рис. 2.6

В этом случае интерполяционный многочлен (10) примет вид

$$P_1(x) = \frac{x-x_1}{x_0-x_1} f(x_0) + \frac{x-x_0}{x_1-x_0} f(x_1). \quad (11)$$

При  $x=x_0$  получим

$$P_1(x_0) = \frac{x_0-x_1}{x_0-x_1} f(x_0) + \frac{x_0-x_0}{x_1-x_0} f(x_1) = f(x_0),$$

а при  $x=x_1$

$$P_1(x_1) = \frac{x_1-x_1}{x_0-x_1} f(x_0) + \frac{x_1-x_0}{x_1-x_0} f(x_1) = f(x_1).$$

Геометрический смысл такого интерполирования очевиден: график функции  $y=P_1(x)$  есть прямая, проходящая через точки с координатами  $(x_0; f(x_0))$  и  $(x_1; f(x_1))$  (рис. 2.5). Чтобы вычислить теперь приближенное значение функции  $f(x)$  в точке  $x^* \in [x_0, x_1]$ , можно вычислить значение  $y=P_1(x^*)$ , подставив  $x^*$  в формулу (11).

В случае  $m=2$  интерполирование носит название *квадратичного*, так как получается интерполяционный многочлен не выше 2-й степени. График функции  $y=P_2(x)$  — парабола, проходящая через точки

$(x_0; f(x_0)), (x_1; f(x_1)), (x_2; f(x_2))$  (или прямая, если многочлен окажется 1-й степени и 0-й) (рис. 2.6).

**З а м е ч а н и я.**

1. При построении интерполяционного многочлена Лагранжа мы не налагаем никаких ограничений на расположение узлов интерполяции на отрезке  $[a, b]$ . Чаще всего узлы на отрезке интерполяции располагают равномерно.

2. Если построен интерполяционный многочлен Лагранжа  $P_m(x)$  по  $(m+1)$ -му узлу, то добавление еще одного узла потребует пересчета всех слагаемых в многочлене (10) и добавления еще одного слагаемого.

3. Естественно было бы ожидать, что чем больше узлов интерполяции для приближения функции на заданном отрезке мы возьмем, т. е. чем большей степени будет интерполяционный многочлен, тем лучше он будет приближать функцию. Разумеется, во всех узлах интерполяции многочлен и функция будут совпадать. Но будет ли уменьшаться в остальных точках отрезка интерполяции модуль разности между значением функции и значением приближающего ее многочлена с ростом количества узлов? Оказывается, это не всегда так. Например, для такой функции, как  $y = 1/(1+25x^2)$ , на отрезке  $[-1, 1]$  с ростом количества узлов интерполяции  $n$  максимум модуля разности между значением функции и значением интерполяционного многочлена неограниченно возрастает, что можно записать следующим образом:

$$\lim_{n \rightarrow \infty} \max_{x \in [-1, 1]} |f(x) - P_n(x)| = \infty.$$

Таким образом, рост количества узлов интерполяции на заданном отрезке еще не гарантирует, что во всех точках этого отрезка будет уменьшаться модуль разности между значениями функции и приближающими ее интерполяционными многочленами.

**П р и м е р ы.** Построить интерполяционный многочлен Лагранжа по следующей таблице:

1)	$x_i$	1	2
	$f(x_i)$	3	5

$$P_1(x) = \frac{x-2}{1-2} \cdot 3 + \frac{x-1}{2-1} \cdot 5 = -3x + 6 + 5x - 5 = 2x + 1.$$

2)	$x_i$	1	2	4
	$f(x_i)$	0	3	15

$$\begin{aligned} P_2(x) &= \frac{(x-2)(x-4)}{(1-2)(1-4)} \cdot 0 + \frac{(x-1)(x-4)}{(2-1)(2-4)} \cdot 3 + \frac{(x-1)(x-2)}{(4-1)(4-2)} \cdot 15 = \\ &= \frac{x^2 - 5x + 4}{-2} \cdot 3 + \frac{x^2 - 3x + 2}{3 \cdot 2} \cdot 15 = x^2 - 1. \end{aligned}$$

$$3) \begin{array}{c|c|c|c} x_i & 1 & 2 & 3 \\ \hline f(x_i) & 1 & 2 & 3 \end{array}$$

$$P_2(x) = \frac{(x-2)(x-3)}{(1-2)(1-3)} \cdot 1 + \frac{(x-1)(x-3)}{(2-1)(2-3)} \cdot 2 + \frac{(x-1)(x-2)}{(3-1)(3-2)} \cdot 3 = x.$$

Результат последнего примера иллюстрирует замечание, что интерполяционный многочлен, построенный по  $(m+1)$ -й точке, может иметь степень и меньшую, чем  $m$ . В данном случае интерполяционный многочлен, построенный по трем точкам, имеет первую степень.

Предлагаем читателю построить интерполяционные многочлены Лагранжа, используя таблицы значения функции:

$$1) \begin{array}{c|c|c} x_i & 1 & 2 \\ \hline f(x_i) & 2 & 5 \end{array}$$

$$\text{Ответ: } P_1(x) = 3x - 1.$$

$$2) \begin{array}{c|c|c|c} x_i & -1 & 1 & 2 \\ \hline f(x_i) & 1,5 & 1,5 & 3 \end{array}$$

$$\text{Ответ: } P_2(x) = \frac{1}{2}x^2 + 1.$$

$$3) \begin{array}{c|c|c|c|c} x_i & -1 & 0 & 0,5 & 1 \\ \hline f(x_i) & 1 & 1 & 1,375 & 3 \end{array}$$

$$\text{Ответ: } P_3(x) = x^3 + x^2 + 1.$$

## § 2.4. Метод наименьших квадратов

Как мы узнали из предыдущего параграфа, часто возникает необходимость представить функцию, заданную таблицей своих значений в конечном числе точек, в виде некоторой аналитической зависимости. В качестве одного из возможных представлений был предложен интерполяционный многочлен. Но было отмечено, что он не всегда удобен для использования. Во-первых, запись его довольно громоздка, во-вторых, между узлами интерполяции его значения могут сильно отклоняться от значений интерполируемой функции. Наконец, если таблица функции отражает результаты какого-то эксперимента и результаты эти получены в результате измерений, сделанных с некоторой погрешностью, то ошибки измерений неминуемо найдут свое отражение и в построенном интерполяционном многочлене.

Поэтому на практике часто используют другой, в каком-то смысле более удобный и точный способ приближения таблично заданной



функции, называемый методом наименьших квадратов. Его идея состоит в том, чтобы построить функцию, которая как бы «сглаживает» те отклонения, которые обусловлены ошибками измерений. График такой функции вовсе не обязан проходить через точки, заданные таблицей, но и нигде не должен сильно от них отклоняться (рис. 2.7).

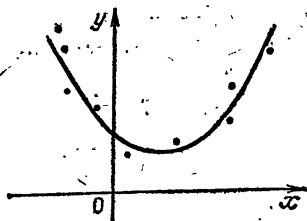


Рис. 2.7

Итак, пусть некоторая функция задана таблицей своих значений:

$x$	$x_1$	$x_2$	$\dots$	$x_n$
$y$	$y_1$	$y_2$	$\dots$	$y_n$

Мы будем искать такую функцию  $\tilde{y} = f(x)$ , чтобы значение

$$\sum_{i=1}^n (y_i - \tilde{y}_i)^2 = (y_1 - \tilde{y}_1)^2 + (y_2 - \tilde{y}_2)^2 + \dots + (y_n - \tilde{y}_n)^2 \quad (12)$$

было минимальным. Для  $f(x)$  можно выбирать, вообще говоря, различные типы функций. Но мы, как и прежде, будем в качестве приближающих функций рассматривать многочлены. И для простоты ограничимся случаем, когда  $f(x)$  будет квадратным трехчленом, т. е.

$$\tilde{y} = a_1 x^2 + a_2 x + a_3. \quad (13)$$

Итак, мы должны найти такой многочлен (13) (т. е. отыскать коэффициенты  $a_1, a_2, a_3$  этого многочлена), чтобы значение выражения (12) было минимальным. Иначе говоря, должна быть минимальной сумма

$$\begin{aligned} S &= \sum_{i=1}^n (y_i - a_1 x_i^2 - a_2 x_i - a_3)^2 = \\ &= (y_1 - a_1 x_1^2 - a_2 x_1 - a_3)^2 + (y_2 - a_1 x_2^2 - a_2 x_2 - a_3)^2 + \dots \\ &\quad \dots + (y_n - a_1 x_n^2 - a_2 x_n - a_3)^2. \end{aligned} \quad (14)$$

Как известно, для функции, рассматриваемой на всей числовой прямой, в точке минимума функции ее производная равна нулю. По условию задачи значения  $x_i$  и  $y_i$  есть заданные постоянные величины.

Рассмотрим сначала  $S$  как функцию только от одной переменной  $a_1$ . Тогда ее производная по  $a_1$  должна быть равна нулю, т. е., дифференцируя выражение (14) по  $a_1$ , получим

$$\begin{aligned} 2(y_1 - a_1 x_1^2 - a_2 x_1 - a_3)(-x_1^2) + 2(y_2 - a_1 x_2^2 - a_2 x_2 - a_3)(-x_2^2) + \dots \\ \dots + 2(y_n - a_1 x_n^2 - a_2 x_n - a_3)(-x_n^2) = 0. \end{aligned}$$

Отсюда

$$\begin{aligned} (a_1 x_1^4 + a_1 x_2^4 + \dots + a_1 x_n^4) + (a_2 x_1^3 + a_2 x_2^3 + \dots + a_2 x_n^3) + \\ + (a_3 x_1^2 + a_3 x_2^2 + \dots + a_3 x_n^2) = y_1 x_1^2 + y_2 x_2^2 + \dots + y_n x_n^2, \end{aligned}$$

или

$$\left(\sum_{i=1}^n x_i^4\right) a_1 + \left(\sum_{i=1}^n x_i^3\right) a_2 + \left(\sum_{i=1}^n x_i^2\right) a_3 = \sum_{i=1}^n y_i x_i^3. \quad (15)$$

Точно так же можно продифференцировать выражение (14) по  $a_2$ , рассматривая это выражение как функцию от  $a_2$ . В точке минимума производная от  $S$  по  $a_2$  должна быть равна нулю:

$$2(y_1 - a_1 x_1^2 - a_2 x_1 - a_3)(-x_1) + 2(y_2 - a_1 x_2^2 - a_2 x_2 - a_3)(-x_2) + \dots \\ \dots + 2(y_n - a_1 x_n^2 - a_2 x_n - a_3)(-x_n) = 0.$$

После преобразования получим

$$(a_1 x_1^3 + a_1 x_2^3 + \dots + a_1 x_n^3) + (a_2 x_1^2 + a_2 x_2^2 + \dots + a_2 x_n^2) + \\ + (a_3 x_1 + a_3 x_2 + \dots + a_3 x_n) = y_1 x_1 + y_2 x_2 + \dots + y_n x_n,$$

или

$$\left(\sum_{i=1}^n x_i^3\right) a_1 + \left(\sum_{i=1}^n x_i^2\right) a_2 + \left(\sum_{i=1}^n x_i\right) a_3 = \sum_{i=1}^n y_i x_i. \quad (16)$$

Наконец, продифференцировав выражение (14) по  $a_3$  и приравняв производную нулю, получим

$$2(y_1 - a_1 x_1^2 - a_2 x_1 - a_3)(-1) + 2(y_2 - a_1 x_2^2 - a_2 x_2 - a_3)(-1) + \dots \\ \dots + 2(y_n - a_1 x_n^2 - a_2 x_n - a_3)(-1) = 0.$$

Отсюда

$$(a_1 x_1^2 + a_1 x_2^2 + \dots + a_1 x_n^2) + (a_2 x_1 + a_2 x_2 + \dots + a_2 x_n) + \\ + \underbrace{(a_3 + a_3 + \dots + a_3)}_n = y_1 + y_2 + \dots + y_n,$$

или

$$\left(\sum_{i=1}^n x_i^2\right) a_1 + \left(\sum_{i=1}^n x_i\right) a_2 + n a_3 = \sum_{i=1}^n y_i. \quad (17)$$

Объединив (15), (16) и (17), получим систему трех алгебраических уравнений с тремя неизвестными  $a_1$ ,  $a_2$  и  $a_3$ :

$$\begin{cases} \left(\sum_{i=1}^n x_i^4\right) a_1 + \left(\sum_{i=1}^n x_i^3\right) a_2 + \left(\sum_{i=1}^n x_i^2\right) a_3 = \sum_{i=1}^n y_i x_i^3, \\ \left(\sum_{i=1}^n x_i^3\right) a_1 + \left(\sum_{i=1}^n x_i^2\right) a_2 + \left(\sum_{i=1}^n x_i\right) a_3 = \sum_{i=1}^n y_i x_i, \\ \left(\sum_{i=1}^n x_i^2\right) a_1 + \left(\sum_{i=1}^n x_i\right) a_2 + n a_3 = \sum_{i=1}^n y_i. \end{cases} \quad (18)$$

В системе (18) коэффициенты при неизвестных  $a_1$ ,  $a_2$  и  $a_3$  легко вычислить, так как все значения  $x_i$  и  $y_i$  ( $i=1, 2, \dots, n$ ) известны. Дальше остается решить систему трех уравнений с тремя неизвестными. Найденные значения  $a_1$ ,  $a_2$ ,  $a_3$  и определяют искомую функцию

$$\tilde{y} = a_1 x^2 + a_2 x + a_3.$$

Построим, например, методом наименьших квадратов многочлен второй степени, приближающий таблично заданную функцию. Для проверки метода можно взять в качестве исходных значений приближаемой функции несколько точек на параболe, заданной некоторым конкретным квадратным трехчленом. Естественно, в результате работы по методу наименьших квадратов мы должны получить квадратный трехчлен с этими же коэффициентами:

$x$	$-2$	$-1$	$0$	$2$	$3$
$y$	$4$	$0$	$-2$	$0$	$4$

Нетрудно видеть, что все эти точки лежат на параболe  $y = x^2 - x - 2$ .

Вычислим коэффициенты в системе (18):

$$\sum_{i=1}^5 x_i = -2 - 1 + 0 + 2 + 3 = 2,$$

$$\sum_{i=1}^5 x_i^2 = 4 + 1 + 0 + 4 + 9 = 18,$$

$$\sum_{i=1}^5 x_i^3 = -8 - 1 + 0 + 8 + 27 = 26,$$

$$\sum_{i=1}^5 x_i^4 = 16 + 1 + 0 + 16 + 81 = 114,$$

$$\sum_{i=1}^5 y_i = 4 + 0 - 2 + 0 + 4 = 6,$$

$$\sum_{i=1}^5 y_i x_i = -8 + 0 + 0 + 0 + 12 = 4,$$

$$\sum_{i=1}^5 y_i x_i^2 = 16 + 0 + 0 + 0 + 36 = 52.$$

В нашем примере система (18) примет вид

$$\begin{cases} 114a_1 + 26a_2 + 18a_3 = 52, \\ 26a_1 + 18a_2 + 2a_3 = 4, \\ 18a_1 + 2a_2 + 5a_3 = 6. \end{cases}$$

Решив методом исключения эту систему, получим  $a_1 = 1$ ,  $a_2 = -1$ ,  $a_3 = -2$ , т. е.  $\tilde{y} = x^2 - x - 2$ , что и требовалось.

Аналогично можно отыскивать в качестве приближающей функции и многочлен любой другой степени. Так, построим методом наименьших квадратов многочлен третьей степени, приближающий функцию теплоемкости воды, заданную таблицей значений. Эта таблица получена из эксперимента и отображает зависимость теплоемкости от температуры, причем теплоемкость при  $15^\circ\text{C}$  принята за единицу.

$T, ^\circ\text{C}$	$C_p$	$T, ^\circ\text{C}$	$C_p$	$T, ^\circ\text{C}$	$C_p$
0	1,00762	35	0,99818	70	1,00091
5	1,00392	40	0,99828	75	1,00167
10	1,00153	45	0,99837	80	1,00253
15	1,00000	50	0,99849	85	1,00351
20	0,99907	55	0,99919	90	1,00461
25	0,99852	60	0,99967	95	1,00586
30	0,99826	65	1,00024	100	1,00721

Для отыскания коэффициентов искомого многочлена третьей степени  $a_1x^3 + a_2x^2 + a_3x + a_4$  здесь пришлось составить и решить систему четырех линейных алгебраических уравнений с четырьмя неизвестными  $a_1, a_2, a_3, a_4$ . Очевидно, в этом примере вычисления были весьма трудоемкими, и вряд ли здесь можно было обойтись без вычислительных средств.

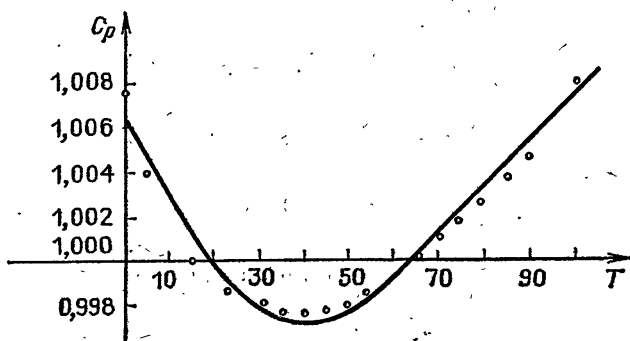


Рис. 2.8

График многочлена вместе с точками таблицы представлен на рис. 2.8. Из рисунка видно, что построенная кубическая парабола является хорошим приближением к заданной функции.

В заключение параграфа дадим читателю возможность проверить, как он усвоил метод наименьших квадратов, для чего предлагаем выполнить несколько упражнений.

1) Построить методом наименьших квадратов многочлен 2-й степени, приближающий следующую таблично заданную функцию:

а)

$x$	-1	0	1	2	3
$y$	4	0	-2	-2	0

6)

$x$	-4	-3	-1	0	1	2
$y$	12	-1	-7	-7	1	9

2) Доказать, что если с помощью метода наименьших квадратов искать уравнение приближающей функции в виде  $y=kx+b$ , причем исходная функция задана таблицей, содержащей лишь две точки  $(x_1, y_1)$  и  $(x_2, y_2)$ , то решение системы уравнений для отыскания коэффициентов уравнения искомой кривой приведет к уравнению прямой, проходящей через эти точки.

3) Пусть таблица приближаемой функции содержит две точки  $(x_1, y_1)$  и  $(x_2, y_2)$ . Доказать, что метод наименьших квадратов для отыскания приближающей функции вида  $\tilde{y}=a_1x^2+a_2x+a_3$  даст бесконечное множество решений. Проверить этот факт на примере функции

$x$	2	4
$y$	-1	3

## § 2.5. Приближенное решение уравнения $f(x)=0$

В этой главе мы опишем некоторые простейшие методы приближенного вычисления корней уравнения  $f(x)=0$ , где  $f(x)$ —заданная функция.

Корнем уравнения  $f(x)=0$  называют число  $x^*$ , которому соответствует число 0 в области значений  $f(x)$ , т. е.  $f(x^*)=0$ .

Найти точные значения корней уравнения удастся только в исключительных случаях, обычно когда есть формула для корней, выражающая их через известные величины. Однако даже для алгебраических уравнений, т. е. уравнений, в которых  $f(x)$  является произвольным алгебраическим многочленом степени  $n$ , корни могут быть найдены через коэффициенты многочлена с помощью формул, содержащих арифметические операции и операцию извлечения корня, только в случае, когда степень многочлена не превосходит четырех. Поэтому большое значение имеют методы приближенного вычисления корней уравнения  $f(x)=0$ .

Будем предполагать, что  $f(x)$ —функция, непрерывная на некотором отрезке  $[a, b]$ , т. е. график функции—непрерывная линия, которую можно нарисовать, не отрывая карандаш от бумаги.

Пусть, кроме того, функция на концах отрезка  $[a, b]$  принимает значения разных знаков. Тогда найдется хотя бы одна точка, в которой график функции пересечет ось  $x$ . Каждая такая точка является корнем уравнения  $f(x)=0$ , так как в ней функция  $f(x)$  обращается в нуль.

Так, если нам нужно оказаться на другой стороне реки, мы хотя бы один раз должны ее пересечь или же это нужно сделать нечетное число раз (рис. 2.9).

Предположим далее, что нам уже заранее известны интервалы, в которых заключены корни уравнения  $f(x)=0$ , при этом в каждом интервале имеется только один корень. Такое разделение корней можно выполнить, например, графически. Разумеется, построив график функции  $y=f(x)$ , мы сможем лишь примерно указать корни уравнения или интервалы, в которых содержится по одному корню уравнения. Дальнейшая задача заключается в вычислении нужного нам корня с необходимой точностью. Описанные ниже методы и предназначены для решения этой задачи.

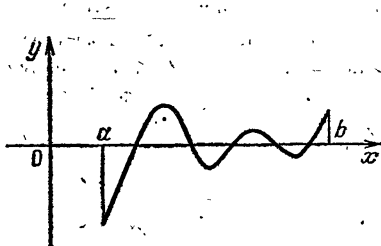


Рис. 2.9

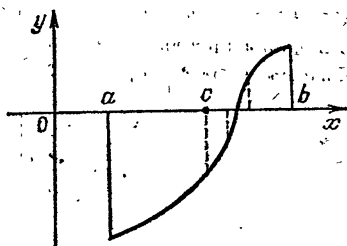


Рис. 2.10

**Метод деления отрезка пополам.** Пусть  $f(x)$  — непрерывная функция на отрезке  $[a, b]$  и известно, что уравнение  $f(x)=0$  имеет на этом отрезке единственный корень  $x^*$ , который нам пока неизвестен. Наша задача состоит в достаточно точном нахождении значения  $x^*$ .

Предположим для определенности, что  $f(a) < 0$ ,  $f(b) > 0$ . Для нахождения корня поступаем так: разделим отрезок  $[a, b]$  пополам. Пусть точкой деления будет  $c$ . Вычисляем  $f(c)$ . Может случиться, что  $f(c)=0$ . Тогда  $c$  — искомый корень уравнения, т. е.  $x^*=c$ .

Если  $f(c) \neq 0$ , то смотрим на знак  $f(c)$ . Если  $f(c) < 0$ , то корень должен лежать на отрезке  $[c, b]$  (как на рис. 2.10); если же  $f(c) > 0$ , то корень — на отрезке  $[a, c]$ . Выбираем из этих отрезков тот, на котором лежит корень, и обозначим его  $[a_1, b_1]$ . Длина отрезка  $[a_1, b_1]$  вдвое меньше длины исходного отрезка  $[a, b]$ , на концах его функция  $y=f(x)$  имеет разные знаки, следовательно, на нем находится искомый корень нашего уравнения. Делим теперь отрезок  $[a_1, b_1]$  пополам. Опять берем точку деления  $c_1$ . Если  $f(c_1)=0$ , то корень найден:  $x^*=c_1$ . Если же  $f(c_1) \neq 0$ , то из двух отрезков  $[a_1, c_1]$  и  $[c_1, b_1]$  выбираем тот, на концах которого функция  $f(x)$  имеет разные знаки. Этот отрезок обозначаем  $[a_2, b_2]$ . Его длина уже вдвое меньше, чем длина  $[a_1, b_1]$ , и вчетверо меньше, чем длина  $[a, b]$ . Делим пополам отрезок  $[a_2, b_2]$ . И так далее. Процесс может оборваться, если в какой-то точке деления  $c_i$  окажется, что  $f(c_i)=0$ . Тогда  $c_i$  — искомый

корень  $x^*$ . Если же процесс продолжается неограниченно, то получается последовательность вложенных друг в друга отрезков, где длина каждого последующего отрезка вдвое меньше предыдущего.

Легко видеть, что  $b_n - a_n = (b - a)/2^n$ . Очевидно, что при неограниченно возрастающем  $n$  эта величина становится сколь угодно малой, т. е. длины отрезков  $[a_n, b_n]$  стремятся к нулю. По построению корень уравнения  $x^*$  находится внутри каждого такого отрезка, и, следовательно, расстояние между  $x^*$  и любым из концов отрезка  $[a_n, b_n]$  меньше длины этого отрезка. Поэтому, если мы хотим, чтобы ошибка в определении корня уравнения не превышала некоторого заданного малого значения  $\varepsilon > 0$ , надо продолжать процесс до тех пор, пока длина отрезка, содержащего корень  $x^*$ , не станет меньше  $\varepsilon$ . Тогда оба конца такого отрезка  $[a_n, b_n]$  будут отстоять от искомого корня  $x^*$  на расстояние, меньшее чем  $\varepsilon$ , и любой из этих концов можно принять за приближенное значение корня.

**Пример.** Применим изложенный метод к решению уравнения  $f(x) = x^2 - 2 = 0$ , т. е. по существу вычислим приближенно значение  $\sqrt{2}$ . Корень уравнения будем искать на отрезке  $[1, 2]$ , так как  $f(1) < 0$ ,  $f(2) > 0$ . Пусть точность, с которой необходимо выполнить вычисления,  $\varepsilon = 0,001$ . Составим таблицу вычислений.

$[a_i, b_i]$	$b_i - a_i$	$c_i = (a_i + b_i)/2$	$f(c_i) \gtrless 0$
$[a_0, b_0] \rightarrow [1, 2]$	1	$3/2 = 1,5$	$1/4 > 0$
$[a_1, b_1] \rightarrow [1, 3/2]$	$1/2$	$5/4 = 1,25$	$-7/16 < 0$
$[a_2, b_2] \rightarrow [5/4, 3/2]$	$1/4$	$11/8 = 1,375$	$-7/64 < 0$
$[a_3, b_3] \rightarrow [11/8, 3/2]$	$1/8$	$23/16 = 1,4375$	$17/256 > 0$
$[a_4, b_4] \rightarrow [11/8, 23/16]$	$1/16$	$45/32 = 1,40635$	$-23/1024 < 0$
$[a_5, b_5] \rightarrow [45/32, 23/16]$	$1/32$	$91/64 = 1,4219$	$89/4096 > 0$
$[a_6, b_6] \rightarrow [45/32, 91/64]$	$1/64$	$181/128 = 1,4141$	$-7/16384 < 0$
$[a_7, b_7] \rightarrow [181/128, 91/64]$	$1/128$	$363/256 = 1,4180$	$697/65536 > 0$
$[a_8, b_8] \rightarrow [181/128, 363/256]$	$1/256$	$725/512 = 1,4160$	$1337/262144 > 0$
$[a_9, b_9] \rightarrow [181/128, 725/512]$	$1/512$	$1449/1024 = 1,4150$	$2449/1048576 > 0$
$[a_{10}, b_{10}] \rightarrow [181/128, 1449/1024]$	$1/1024$	$2897/2048 = 1,4146$	$4001/4194304 > 0$

Мы закончили вычисления тогда, когда достигнута заданная точность вычислений, т. е.  $b_n - a_n < \varepsilon$ . В нашем случае  $b_{10} - a_{10} =$

$= 1/1024 < 0,001$ , и, значит, любое из двух чисел  $a_{10}$  или  $b_{10}$  можно принять за приближенное значение корня:

$$a_{10} = 181/128 = 1,4141, \quad b_{10} = 1449/1024 = 1,4150.$$

Мы также получили и следующее приближение к корню — средняя точка этого отрезка  $c = 1,4146$ .

На этом простом примере мы убедились, что даже для решения простой задачи потребовалось проделать много трудоемких и однообразных вычислений. Их было бы еще больше, если бы усложнился вид функции  $f(x)$  и понадобилось бы увеличить точность результата, т. е. уменьшить  $\varepsilon$ . Поэтому, конечно же, решая подобные задачи, необходимо призывать на помощь вычислительную технику, которая позволит быстро, надежно и с высокой точностью получить искомый результат.

**Метод касательных (метод Ньютона).** Пусть уравнение  $f(x) = 0$  имеет единственный корень  $x^*$  на отрезке  $[a, b]$ . Предположим далее, что  $f(x)$  непрерывна на  $[a, b]$  и всюду на этом отрезке имеет производную  $f'(x)$ , отличную от нуля. Последнее условие означает, что функция  $y = f(x)$  на отрезке  $[a, b]$  строго монотонна — либо возрастает, либо убывает. Пусть график функции  $y = f(x)$  имеет в окрестности корня вид, показанный на рис. 2.11. Будем искать приближенное значение корня уравнения  $f(x) = 0$  методом, который называется методом касательных, или методом Ньютона.

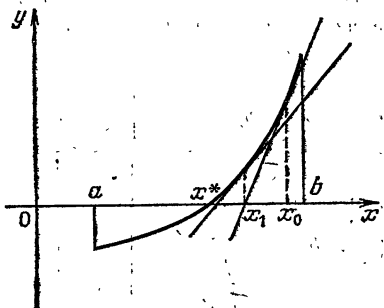


Рис. 2.11

Метод касательных имеет очень простой геометрический смысл. Пусть малый интервал, внутри которого есть корень, уже выделен. И пусть на этом интервале функция выпукла вверх или вниз. Возьмем произвольную точку  $x_0$  вблизи корня. Проведем в точке  $(x_0, f(x_0))$  касательную к графику функции  $y = f(x)$ . Уравнение этой касательной:

$$y = f(x_0) + f'(x_0)(x - x_0). \quad (19)$$

Можно ожидать, что точка пересечения  $x_1$  касательной с осью  $x$  будет расположена ближе к корню, чем точка  $x_0$ . Найдём точку  $x_1$  из уравнения (19). В точке пересечения графика касательной с осью  $x$  ордината касательной  $y$  равна 0:

$$0 = f(x_0) + f'(x_0)(x_1 - x_0).$$

Отсюда

$$x_1 = x_0 - f(x_0)/f'(x_0).$$



Определив точку  $x_1$ , проведем касательную к графику функции  $y=f(x)$  в точке  $(x_1, f(x_1))$  и найдем точку ее пересечения с осью абсцисс  $x_2$ . Так же, как и в предыдущем случае, получим

$$x_2 = x_1 - f(x_1)/f'(x_1).$$

После  $n$ -кратного повторения этой процедуры имеем

$$x_n = x_{n-1} - f(x_{n-1})/f'(x_{n-1}). \quad (20)$$

Получим последовательность  $x_0, x_1, \dots, x_n, \dots$ , которая неограниченно приближается к корню  $x^*$  уравнения  $f(x)=0$ . Такие формулы, в которых  $n$ -й член последовательности можно выразить через значения ее первых  $n-1$  членов, называются рекуррентными. В нашем случае каждый член последовательности зависит только от значения предыдущего члена этой последовательности. Рекуррентные формулы очень удобны для расчетов на ЭВМ.

Заметим, что если начальное приближение  $x_0$  выбрано далеко от корня (на рис. 2.11 — точка  $a$ ), то может случиться, что первый же шаг процесса дает касательную, которая пересечет ось  $x$  вне отрезка  $[a, b]$ . В этом случае процесс оборвется. Поэтому так важно отделить, или, как говорят, локализовать корень, с тем чтобы начальное приближение  $x_0$  оказалось достаточно близко от корня. На практике часто поступают следующим образом. Выбирают начальное приближение, пользуясь графиком функции, или же делают несколько шагов методом деления отрезка пополам. Затем задаются некоторым малым числом  $\varepsilon > 0$ , характеризующим точность вычислений. Если при некотором  $n$  будет достигнуто неравенство  $|x_{n+1} - x_n| < \varepsilon$ , то считают, что любое из этих чисел  $x_n$  или  $x_{n+1}$  — хорошее приближение к корню  $x^*$ . Если же в процессе вычислений разность  $|x_{n+1} - x_n|$  остаётся большой, то ищут другое начальное приближение.

В методе касательных, в отличие от метода деления отрезка пополам, приходится кроме значения самой функции  $f(x)$  вычислять еще и производную  $f'(x)$ . Но эти затраты окупаются высокой скоростью, с которой последовательность точек  $\{x_i\}$  ( $i=1, 2, \dots, n$ ) приближается к точному значению корня  $x^*$ . Заметим, что все точки  $x_i$  находятся по одну сторону от корня уравнения (мы предполагаем, что кривая выпукла в окрестности корня) и из неравенства  $|x_{n+1} - x_n| < \varepsilon$ , вообще говоря, еще не следует, что  $|x_{n+1} - x^*| < \varepsilon$ . (Вспомним, что в методе деления отрезка пополам из того, что разность двух последовательных приближений была меньше некоторого  $\varepsilon > 0$ , следовало, что эти приближения отличаются от точного значения корня  $x^*$  меньше, чем на то же значение  $\varepsilon$ .)

**Пример.** Воспользуемся методом касательных для извлечения квадратного корня из произвольного положительного числа  $a$ .

Нам надо решить уравнение

$$f(x) = x^2 - a = 0.$$

Подставим функцию  $f(x)$  в формулу (20):

$$x_n = x_{n-1} - (x_{n-1}^2 - a) / (2x_{n-1}) \quad (21)$$

(так как  $f'(x) = 2x$ ).

Преобразуем формулу (21):

$$x_n = \frac{1}{2} \left( x_{n-1} + \frac{a}{x_{n-1}} \right). \quad (22)$$

Для иллюстрации метода вычислим с помощью рекуррентной формулы (22) приближенное значение  $\sqrt{2}$ . Корень будем искать на отрезке  $[1, 2]$ . На этом отрезке, очевидно,  $f'(x) > 0$ , т. е. функция  $f(x) = x^2 - 2$  монотонно возрастает,  $f(1) = -1$ ;  $f(2) = 2$ . За нулевое приближение примем  $x_0 = 1$ . Тогда

$$x_1 = \frac{1}{2} \left( x_0 + \frac{a}{x_0} \right) = \frac{1}{2} \left( 1 + \frac{2}{1} \right) = 3/2 = 1,5,$$

$$x_2 = \frac{1}{2} \left( 1,5 + \frac{2}{1,5} \right) = 1,41666 \dots,$$

$$x_3 = \frac{1}{2} \left( 1,41666 + \frac{2}{1,41666} \right) = 1,414215 \dots,$$

$$x_4 = \frac{1}{2} \left( 1,414215 + \frac{2}{1,414215} \right) = 1,414213 \dots$$

Как известно, значение  $\sqrt{2} = 1,414213 \dots$ . Мы получим хорошее приближение к нему очень быстро: уже на 3-м шаге ошибка только в 6-м знаке после запятой. А разность 3-го и 4-го приближений очень мала:

$$|x_3 - x_4| = 1,414215 - 1,414213 = 0,000002.$$

Напомним, что при решении этой задачи методом последовательного деления отрезка пополам нам для достижения точности результата  $\varepsilon = 0,001$  понадобилось 10 шагов.

**Метод хорд.** По-прежнему будем решать уравнение  $f(x) = 0$ , которое имеет корень  $x^*$  на отрезке  $[a, b]$ ,  $f'(x) \neq 0$  на  $[a, b]$ .

Теперь мы рассмотрим метод хорд, который, как и метод касательных, имеет наглядный геометрический смысл. Так же, как и в методе касательных, будем предполагать, что функция  $y = f(x)$  в достаточно малой окрестности корня выпукла вверх или вниз.

Пусть, например, график функции  $y = f(x)$  в окрестности корня имеет вид, показанный на рис. 2.12. Выберем точки  $\bar{x}$  и  $x_0$ , принад-

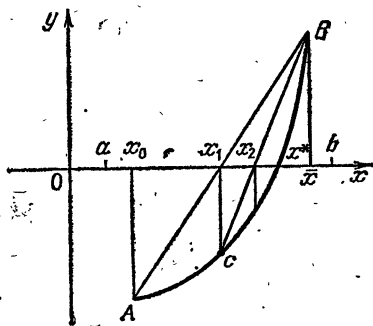


Рис. 2.12

лежащие отрезку  $[a, b]$ , как это показано на рис. 2.12. Проведем хорду, соединяющую точки графика с абсциссами  $x_0$  и  $\bar{x}$ , и найдем точку  $x_1$  ее пересечения с осью абсцисс. Из подобия треугольников  $Ax_0x_1$  и  $B\bar{x}x_1$  получим

$$(x_0 - x_1)/f(x_0) = (\bar{x} - x_1)/f(\bar{x}).$$

Отсюда

$$x_1 = (\bar{x}f(x_0) - x_0f(\bar{x}))/f(x_0) - f(\bar{x}). \quad (23)$$

Затем проводим хорду через точки  $C$  (точка графика функции  $y = f(x)$  с абсциссой  $x_1$ ) и  $B$ , находим точку ее пересечения с осью  $x$ . Получаем

$$x_2 = (\bar{x}f(x_1) - x_1f(\bar{x}))/f(x_1) - f(\bar{x}).$$

Поступая далее подобным образом, видим, что хорда, полученная на  $(n+1)$ -м шаге, пересекает ось абсцисс в точке  $x_{n+1}$ :

$$x_{n+1} = (\bar{x}f(x_n) - x_nf(\bar{x}))/f(x_n) - f(\bar{x}). \quad (24)$$

Получающаяся последовательность  $x_1, x_2, \dots, x_n, \dots$  неограниченно приближается к точному значению  $x^*$  корня уравнения  $f(x) = 0$ .

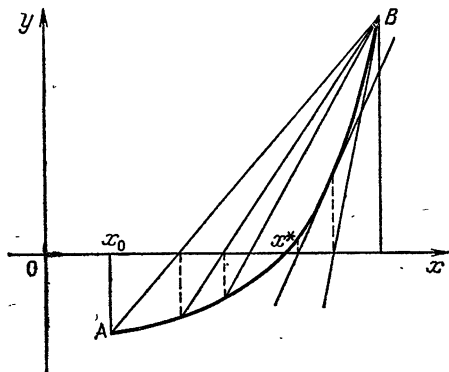


Рис. 2.13

При вычислении корня уравнения  $f(x) = 0$  методом хорд процесс продолжают, как и в методе касательных, до тех пор, пока абсолютная величина разности двух приближений не станет меньше наперед заданного малого положительного числа  $\varepsilon$ :

$$|x_{n+1} - x_n| < \varepsilon. \quad (25)$$

И здесь из этого неравенства, вообще говоря, не следует, что выполнится условие

$$|x^* - x_n| < \varepsilon. \quad (26)$$

Заметим, что как метод хорд, так и метод касательных дает приближение к корню либо все время с недостатком, либо все время с избытком. Причем, если метод касательных дает приближения с избытком, то метод хорд в этом случае дает приближения с недостатком и наоборот. Поэтому, применяя метод хорд совместно с методом касательных, можно получить последовательные приближения к корню с двух сторон (см. рис. 2.13).

Корень уравнения будет находиться между приближениями, одно из которых  $x_n$ , получается методом хорд, другое  $x_{n+1}$  — методом касательных. Поэтому в данном случае из неравенства (25) будет вытекать и неравенство (26), говорящее о том, что мы нашли приближенное значение  $x_n$  корня уравнения  $f(x)=0$ , отличающееся по абсолютной величине от точного значения корня  $x^*$  меньше, чем на  $\varepsilon$ .

**Пример.** Рассмотрим все ту же задачу отыскания, приближенного значения  $\sqrt{2}$ . Будем для этого решать методом хорд уравнение  $f(x)=x^2-2=0$  на отрезке  $[1, 2]$ , где  $f'(x) \neq 0$  (точнее говоря,  $f'(x) > 0$ ,  $f(1)=-1$ ,  $f(2)=2$ ). Зададим  $\varepsilon=0,001$ . Положим  $\bar{x}=2$ ,  $x_0=1$ . Тогда  $f(\bar{x})=f(2)=2^2-2=2$ . По формуле (23) получим

$$x_1 = \frac{\bar{x}f(x_0) - x_0f(\bar{x})}{f(x_0) - f(\bar{x})} = \frac{2 \cdot f(1) - 1 \cdot f(2)}{f(1) - f(2)} = \frac{2(-1) - 1 \cdot 2}{-1 - 2} = \frac{4}{3}.$$

Будем применять формулу (24) последовательно для  $n=1, 2, 3, \dots$  до тех пор, пока не получим, что  $|x_{n+1} - x_n| < \varepsilon$ , т. е.

$$x_2 = \frac{\bar{x}f(x_1) - x_1f(\bar{x})}{f(x_1) - f(\bar{x})} = \frac{2 \cdot f(4/3) - (4/3)f(2)}{f(4/3) - f(2)} = \\ = \frac{-2(16/9 - 2) - (4/3) \cdot 2}{(16/9 - 2) - 2} = \frac{7}{5} = 1,4,$$

$$x_3 = \frac{\bar{x}f(x_2) - x_2f(\bar{x})}{f(x_2) - f(\bar{x})} = \frac{2 \cdot f(7/5) - 7/5 f(2)}{f(7/5) - f(2)} = \\ = \frac{2(49/25 - 2) - 7/5 \cdot 2}{(49/25 - 2) - 2} = \frac{24}{17} = 1,4118,$$

$$x_4 = \frac{\bar{x}f(x_3) - x_3f(\bar{x})}{f(x_3) - f(\bar{x})} = \frac{2 \cdot f(24/17) - (24/17)f(2)}{f(24/17) - f(2)} = \\ = \frac{2 \cdot (576/289 - 2) - (24/17) \cdot 2}{(576/289 - 2) - 2} = \frac{41}{29} = 1,4138,$$

$$x_5 = \frac{\bar{x}f(x_4) - x_4f(\bar{x})}{f(x_4) - f(\bar{x})} = \frac{2 \cdot f(41/29) - (41/29)f(2)}{f(41/29) - f(2)} = \\ = \frac{2 \cdot (1681/841 - 2) - (41/29) \cdot 2}{(1681/841 - 2) - 2} = \frac{2380}{1681} = 1,4141.$$

Как видно,  $|x_5 - x_4| \simeq 0,0003 < 0,001$ . Поэтому в качестве приближенного значения корня уравнения  $x^2-2=0$  можно принять  $x_5 = 1,4141$ . Для его нахождения пришлось 4 раза проделать вычисления по рекуррентной формуле (24).

В этом примере все приближения мы получали с недостатком. При решении этой же задачи методом касательных все приближения получались с избытком.

Предлагаем читателю, используя приближенные методы, решить следующие задачи:

1) Используя метод хорд и метод касательных, найти корень уравнения  $f(x) = x^2 - 6x + 5 = 0$  с точностью  $\varepsilon = 0,001$ .

2) Используя метод касательных, найти корень уравнения  $f(x) = x - \cos x = 0$  при  $x > 0$ ,  $\varepsilon = 0,001$ . Значения  $\cos x$  получать по таблицам или с помощью калькулятора.

3) Методом касательных и методом хорд решить уравнение  $f(x) = x - \ln x - 2 = 0$ . Значения  $\ln 2$  вычислять по таблице или с помощью калькулятора.

## § 2.6. Приближенные формулы для вычисления определенных интегралов

В школьном курсе алгебры определенный интеграл функции  $f(x)$  на отрезке  $[a; b]$  определяется как приращение первообразной этой функции, т. е.

$$\int_a^b f(x) dx = F(b) - F(a), \quad (27)$$

где  $F(x)$  — первообразная функции  $f(x)$ . Это равенство называется формулой Ньютона — Лейбница. Если подынтегральная функция не-

отрицательна, то  $\int_a^b f(x) dx$  имеет наглядный геометрический смысл — он равен площади криволинейной трапеции, ограниченной графиком функции  $y = f(x)$ , осью  $x$  и прямыми  $x = a$ ,  $x = b$  (рис. 2.14).

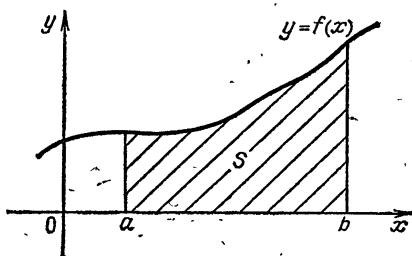


Рис. 2.14

Пользуясь формулой Ньютона — Лейбница, можно вычислять определенный интеграл, если известна первообразная функции  $f(x)$ . Однако на практике часто встречаются случаи, когда первообразная функции  $f(x)$  или не выражается через элементарные функции (на-

пример, для  $f(x) = e^{-x^2}$  или  $f(x) = (\sin x)/x$ , или ее выражение настолько сложно, что воспользоваться им затруднительно. В этих случаях для вычисления определенных интегралов пользуются не формулой Ньютона — Лейбница, а некоторыми формулами, позволяю-

щими вычислить интеграл приближенно, с заданной точностью. Эти приближенные формулы называются квадратурными формулами.

**Формула прямоугольников.** Пусть  $f(x)$  — непрерывная положительная функция на отрезке  $[a, b]$  ( $a < b$ ). Вычислим приближенно площадь криволинейной трапеции, ограниченной графиком функции  $y=f(x)$ , осью  $x$ , прямыми  $x=a$  и  $x=b$  (рис. 2.15). Для этого разобьем отрезок  $[a, b]$  на  $n$  равных отрезков одинаковой длины  $\Delta x = (b-a)/n$ :

$$a=x_0 < x_1 < x_2 < \dots < x_n=b, \\ x_i - x_{i-1} = \Delta x.$$

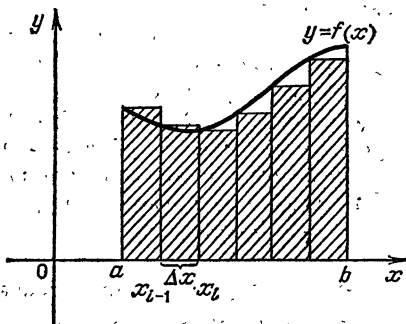


Рис. 2.15

На каждом из отрезков  $[x_{i-1}, x_i]$  как на основании строим прямоугольник с высотой  $f(x_{i-1})$ . Площадь такого прямоугольника равна  $f(x_{i-1}) \cdot (x_i - x_{i-1}) = f(x_{i-1}) \cdot \Delta x$ .

Сумма площадей всех прямоугольников равна

$$f(x_0) \Delta x + f(x_1) \Delta x + \dots + f(x_{n-1}) \Delta x = (f(x_0) + f(x_1) + \dots + f(x_{n-1})) \Delta x = (f(x_0) + f(x_1) + \dots + f(x_{n-1})) (b-a)/n.$$

Если  $n$  — количество точек разбиения отрезка  $[a, b]$  увеличивать, то длина отрезков  $\Delta x$  будет уменьшаться, т. е. прямоугольники будут сужаться, а количество этих прямоугольников будет расти. Ломаная линия, ограничивающая сверху прямоугольники, будет все больше приближаться к графику функции  $f(x)$ . Тем самым сумма площадей прямоугольников будет становиться все ближе к площади криволинейной трапеции. Поэтому можно считать, что эта сумма при достаточно большом  $n$  дает приближенное значение площади криволинейной трапеции, а следовательно,

$$\int_a^b f(x) dx \approx \frac{b-a}{n} (f(x_0) + f(x_1) + \dots + f(x_{n-1})). \quad (28)$$

Формула (28) носит название квадратурной формулы прямоугольников.

Сколько же надо брать точек  $n$  разбиения отрезка  $[a, b]$ , чтобы получить значение интеграла с требуемой точностью?

Точное значение интеграла нам, естественно, неизвестно. Значит, сравнить полученное приближенное значение с ним мы не можем. Поэтому здесь поступают так же, как в методах решения уравнения  $f(x)=0$ ; — сравнивают два последовательных приближения к результату. А получают эти последовательные приближения обычно так.

Задают некоторое начальное значение числа  $n$  точек разбиения отрезка  $[a, b]$ . Вычисляют приближенное значение  $S_1$  интеграла при разбиении отрезка  $[a, b]$  на  $n$  отрезков. Затем удваивают количество отрезков разбиения, т. е. вместо  $n$  берут  $2n$  отрезков, и соответственно длины отрезков разбиения становятся вдвое меньше  $\frac{\Delta x}{2}$ . Вычисляют

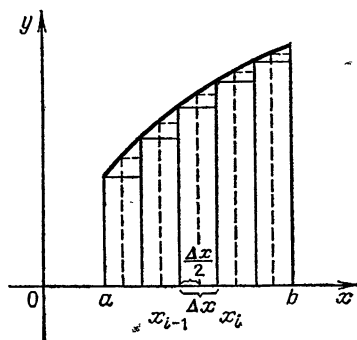


Рис. 2.16

новую сумму площадей более узких прямоугольников  $S_2$  (рис. 2.16). Она точнее приближает искомое значение интеграла. Теперь разность  $|S_2 - S_1|$  сравнивают с наперед заданным малым положительным числом  $\varepsilon$ . Если  $|S_2 - S_1| < \varepsilon$ , то считают, что  $S_2$  можно принять за приближенное значение интеграла, полученное с заданной точностью  $\varepsilon$ .

Если же  $|S_2 - S_1| \geq \varepsilon$ , то еще раз удваивают количество отрезков разбиения и вычисляют новую сумму  $S_3$  площадей прямоугольников. Проверяют, выполнено ли условие  $|S_3 - S_2| < \varepsilon$ . Если оно выполнено, то  $S_3$  — приближенное значение интеграла; если нет, то снова удваивают количество отрезков разбиения и т. д. При увеличении числа  $n$  эти площади все меньше и меньше отличаются друг от друга. Процесс продолжается до тех пор, пока при некотором  $k$ -м разбиении не получится, что  $|S_k - S_{k-1}| < \varepsilon$ .

Для непрерывных функций такой момент обязательно наступит. Тогда  $S_k$  можно принять за приближенное значение интеграла

$$\int_a^b f(x) dx.$$

Нетрудно заметить, что при подсчете суммы  $S_k$  нет нужды вычислять все значения  $f(x_i)$  в точках разбиения. Ведь точки  $x_i$  нового разбиения состоят из точек предыдущего разбиения, в которых значения функции уже вычислены на предыдущем шаге, и точек, являющихся серединами отрезков предыдущего разбиения. Вот в этих средних точках только и нужно вычислять значения функции. А затем их просуммировать и сложить с суммой, полученной от предыдущего разбиения. После чего весь результат умножить на новое значение  $\Delta x$ .

Вычислим, например,  $\int_1^2 x^2 dx$  с точностью  $\varepsilon = 0,02$ . Значение этого

интеграла легко получить по формуле Ньютона — Лейбница:

$$\int_1^2 x^2 dx = \frac{1}{3} x^3 \Big|_1^2 = \frac{1}{3} (8 - 1) = \frac{7}{3} = 2 \frac{1}{3}.$$

Мы испытаем на этом примере формулу прямоугольников. Сначала разобьем отрезок  $[1, 2]$  на два отрезка, затем на четыре, затем на восемь и т. д. Каждую следующую сумму  $S_k$  будем сравнивать с предыдущей  $S_{k-1}$ . Как только условие  $|S_k - S_{k-1}| < \varepsilon$  будет выполнено, значение  $S_k$  примем за приближенное значение интеграла, вычисленное с точностью  $\varepsilon$  (рис. 2.17).

Итак, при  $n=2$  имеем

$$S_1 = \frac{1}{2} \left( f(1) + f\left(\frac{3}{2}\right) \right) = \frac{1}{2} \left( 1 + \frac{9}{4} \right) = \frac{13}{8} = 1 \frac{5}{8} = 1,625,$$

при  $n=4$

$$S_2 = \frac{1}{4} \left( f(1) + \underline{f\left(\frac{5}{4}\right)} + \underline{f\left(\frac{3}{2}\right)} + \underline{f\left(\frac{7}{4}\right)} \right) = \\ = \frac{1}{4} \left( 1 + \frac{25}{16} + \frac{9}{4} + \frac{49}{16} \right) = \frac{63}{32} \approx 1,969.$$

Здесь нам понадобилось вычислить и просуммировать только значения функции в точках  $5/4$  и  $7/4$ :  $\underline{f(5/4)} + \underline{f(7/4)}$  (эти слагаемые у нас подчеркнуты), так как сумма  $f(1) + f(3/2)$  вычислена на предыдущем шаге.

При  $n=8$

$$S_3 = \frac{1}{8} \left( f(1) + \underline{f\left(\frac{9}{8}\right)} + \underline{f\left(\frac{5}{4}\right)} + \right. \\ \left. + \underline{f\left(\frac{11}{8}\right)} + \underline{f\left(\frac{3}{2}\right)} + \underline{f\left(\frac{13}{8}\right)} + \right. \\ \left. + \underline{f\left(\frac{7}{4}\right)} + \underline{f\left(\frac{15}{8}\right)} \right) = \\ = \frac{1}{8} \left( 1 + \frac{81}{64} + \frac{25}{16} + \frac{121}{64} + \frac{9}{4} + \right. \\ \left. + \frac{169}{64} + \frac{49}{16} + \frac{225}{64} \right) = \frac{275}{128} \approx 2,148.$$

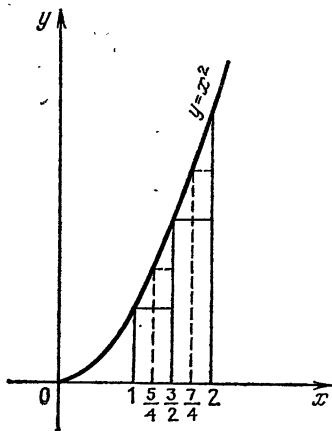


Рис. 2.17

И здесь нам надо вычислить и суммировать лишь половину всех слагаемых (они подчеркнуты), так как остальная сумма вычислена на предыдущем шаге.



При  $n=16$

$$S_4 = \frac{1}{16} \left( f(1) + f\left(\frac{17}{16}\right) + f\left(\frac{9}{8}\right) + f\left(\frac{19}{16}\right) + f\left(\frac{5}{4}\right) + f\left(\frac{21}{16}\right) + f\left(\frac{11}{8}\right) + \right. \\ \left. + f\left(\frac{23}{16}\right) + f\left(\frac{3}{2}\right) + f\left(\frac{25}{16}\right) + f\left(\frac{13}{8}\right) + f\left(\frac{27}{16}\right) + f\left(\frac{7}{4}\right) + f\left(\frac{29}{16}\right) + \right. \\ \left. + f\left(\frac{15}{8}\right) + f\left(\frac{31}{16}\right) \right) = \frac{1}{16} \left( \frac{275}{16} + \frac{289}{256} + \frac{81}{64} + \frac{361}{256} + \frac{25}{16} + \frac{441}{256} + \frac{121}{64} + \right. \\ \left. + \frac{529}{256} + \frac{9}{4} + \frac{625}{256} + \frac{169}{64} + \frac{729}{256} + \frac{49}{16} + \frac{841}{256} + \frac{225}{64} + \frac{361}{256} \right) = \frac{1147}{512} \approx 2,240$$

(первое слагаемое в скобках  $275/16$  — сумма значений функции  $f(x)$ , полученная при  $n=8$ ).

При  $n=32$

$$S_8 = \frac{1}{32} \left( f(1) + f\left(\frac{33}{32}\right) + f\left(\frac{17}{16}\right) + f\left(\frac{35}{32}\right) + f\left(\frac{9}{8}\right) + \right. \\ \left. + f\left(\frac{37}{32}\right) + f\left(\frac{19}{16}\right) + f\left(\frac{39}{32}\right) + f\left(\frac{5}{4}\right) + f\left(\frac{41}{32}\right) + f\left(\frac{21}{16}\right) + \right. \\ \left. + f\left(\frac{43}{32}\right) + f\left(\frac{11}{8}\right) + f\left(\frac{45}{32}\right) + f\left(\frac{23}{16}\right) + f\left(\frac{47}{32}\right) + f\left(\frac{3}{2}\right) + f\left(\frac{49}{32}\right) + \right. \\ \left. + f\left(\frac{25}{16}\right) + f\left(\frac{51}{32}\right) + f\left(\frac{13}{8}\right) + f\left(\frac{53}{32}\right) + f\left(\frac{27}{16}\right) + f\left(\frac{55}{32}\right) + f\left(\frac{7}{4}\right) + \right. \\ \left. + f\left(\frac{57}{32}\right) + f\left(\frac{29}{16}\right) + f\left(\frac{59}{32}\right) + f\left(\frac{15}{8}\right) + f\left(\frac{61}{32}\right) + f\left(\frac{31}{16}\right) + f\left(\frac{63}{32}\right) \right) = \\ = \frac{1}{32} \left( \frac{1147}{32} + \frac{1089}{1024} + \frac{1225}{1024} + \frac{1369}{1024} + \frac{1521}{1024} + \frac{1681}{1024} + \frac{1849}{1024} + \frac{2025}{1024} + \right. \\ \left. + \frac{2209}{1024} + \frac{2401}{1024} + \frac{2601}{1024} + \frac{2809}{1024} + \frac{3025}{1024} + \frac{3249}{1024} + \frac{3481}{1024} + \frac{3721}{1024} + \frac{3969}{1024} \right) = \\ = \frac{1}{32} \cdot \frac{74928}{1024} = \frac{1}{32} \cdot \frac{4683}{64} = 2,287$$

(первое слагаемое в скобках  $1147/32$  — сумма значений функции  $f(x)$ , полученная при  $n=16$ ).

При  $n=64$

$$S_{16} = \frac{1}{64} \left( f(1) + f\left(\frac{65}{64}\right) + f\left(\frac{33}{32}\right) + f\left(\frac{67}{64}\right) + f\left(\frac{17}{16}\right) + \dots \right. \\ \left. \dots + f\left(\frac{31}{16}\right) + f\left(\frac{125}{64}\right) + f\left(\frac{63}{32}\right) + f\left(\frac{127}{64}\right) \right) = \frac{1}{64} \left( \frac{4683}{64} + \frac{1}{2^{12}} (4225 + \right. \\ \left. + 4489 + 4761 + 5041 + 5329 + 5625 + 5929 + 6241 + 6561 + 6889 + \right. \\ \left. + 7225 + 7569 + 7921 + 8281 + 8649 + 9025 + 9409 + 9801 + 1020 + \right. \\ \left. + 10609 + 11025 + 11449 + 11881 + 12321 + 12769 + 13225 + 13689 + \right. \\ \left. + 14161 + 14641 + 15129 + 15625 + 16129) \right) = \frac{1}{64} \left( \frac{4683}{64} + \frac{304624}{2^{12}} \right) = \\ = \frac{1}{64} \left( \frac{299712 + 304624}{4096} \right) = \frac{1}{64} \cdot \frac{604336}{4096} = \frac{1}{64} \cdot \frac{37771}{256} \approx 2,3054$$

(первое слагаемое в скобках 4683/64 — сумма значений функции  $f(x)$ , полученная при  $n=32$ ).

Как видно из вычислений,  $|S_6 - S_5| \approx 0,018 < 0,02$ , т. е. требуемая точность достигнута. Пример показал, что даже для такой простой подынтегральной функции, как  $x^2$ , и столь невысокой точности результата пришлось проделать довольно много трудоемких вычислений.

Следует вообще иметь в виду, что формула (28) часто бывает не очень удобна, так как для получения удовлетворительной точности приходится брать  $n$  очень большим. Это неудобство весьма существенно при ручном счете и даже при счете на микрокалькуляторе. При расчетах на ЭВМ это неудобство уже не является столь существенным.

**Формула трапеций.** Приведем еще одну формулу приближенного вычисления интеграла  $\int_a^b f(x) dx$ . Разобьем промежуток интегрирования

$[a, b]$  как и в случае метода прямоугольников на  $n$  равных частей длины  $\Delta x = (b-a)/n$ :  $a = x_0 < x_1 < x_2 < \dots < x_n = b$ . Из точек деления  $x_i$  проведем прямые, перпендикулярные оси  $x$ . Последовательно соединим точки пересечения этих прямых с графиком функции  $y = f(x)$ . Очевидно, получившаяся ломаная лучше приближает кривую  $y = f(x)$ , чем ломаная в методе прямоугольников. Площадь фигуры, состоящей из  $n$  трапеций, очевидно, близка к площади криволинейной трапеции.

Найдем площадь фигуры, состоящей из трапеций. Площадь трапеции равна высоте трапеции  $(b-a)/n$ , умноженной на длину ее средней линии

$$\frac{f(x_i) + f(x_{i+1})}{2}, \text{ т. е. равна } \frac{b-a}{n} \left( \frac{f(x_i) + f(x_{i+1})}{2} \right).$$

Площадь всей фигуры, очевидно, равна

$$S = \frac{b-a}{n} \left( \frac{f(x_0) + f(x_1)}{2} + \frac{f(x_1) + f(x_2)}{2} + \dots + \frac{f(x_{n-1}) + f(x_n)}{2} \right) = \\ = \frac{b-a}{n} \left( \frac{f(x_0) + f(x_n)}{2} + f(x_1) + f(x_2) + \dots + f(x_{n-1}) \right).$$

Полученная площадь дает приближенное значение [интеграла

$$\int_a^b f(x) dx. \text{ С учетом того, что } f(x_0) = f(a) \text{ и } f(x_n) = f(b), \text{ получаем}$$

$$\int_a^b f(x) dx \approx \frac{b-a}{n} \left( \frac{f(a) + f(b)}{2} + f(x_1) + f(x_2) + \dots + f(x_{n-1}) \right), \quad (29)$$

где  $x_i = a + i \cdot \Delta x = a + i \cdot \frac{b-a}{n}$  ( $i = 1, 2, \dots, n-1$ ).

Формула (29) носит название квадратурной формулы трапеций. Чем больше точек разбиения  $n$  отрезка  $[a, b]$ , тем точнее эта формула.

Для приближенного вычисления интеграла с достаточно высокой точностью поступают так же, как при вычислении с помощью метода прямоугольников, т. е. задают некоторое начальное количество  $n$  точек разбиения отрезка  $[a, b]$ . Вычисляют получившуюся при этом площадь  $S_1$ . Затем удваивают количество точек разбиения и вычисляют площадь  $S_2$ . Сравнивают их по абсолютной величине, и если  $|S_1 - S_2| < \varepsilon$ , где  $\varepsilon > 0$  — заданная точность вычислений, то принимают  $S_2$  в качестве приближенного значения результата. Если же  $|S_1 - S_2| \geq \varepsilon$ , то снова удваивают количество точек разбиения, вычисляют  $S_3$ . Сравнивают  $S_3$  с  $S_2$  и т. д. Процесс продолжается до тех пор, пока при каком-то  $k$ -м удвоении числа разбиений отрезка  $[a, b]$  не получится, что  $|S_k - S_{k-1}| < \varepsilon$ . Тогда  $S_k$  — искомое прибли-

женное значение интеграла  $\int_a^b f(x) dx$ . И опять-таки, так же как в методе прямоугольников, при нахождении  $S_k$  нужно вычислять и суммировать не все  $f(x_i)$  ( $i=1, 2, \dots, n-1$ ), а только значения функции  $f(x)$  в тех точках, которые не совпадают с точками предыдущего разбиения. Вычислив сумму этих значений, ее складывают с суммой значений функции, полученной на предыдущем шаге процесса при вычислении  $S_{k-1}$ , а затем общую сумму умножают на  $(b-a)/n$ .

Для иллюстрации метода снова вычислим приближенное значение интеграла  $\int_1^2 x^2 dx$ , воспользовавшись методом трапеций.

Положим  $\varepsilon = 0,01$ , и пусть начальное значение  $n = 2$ . Тогда

$$S_1 = \frac{1}{2} \left( \frac{f(1) + f(2)}{2} + f\left(\frac{3}{2}\right) \right) = \frac{1}{2} \left( \frac{5}{2} + f\left(\frac{3}{2}\right) \right) = \\ = \frac{1}{2} \left( \frac{5}{2} + \frac{9}{4} \right) = \frac{1}{2} \cdot \frac{19}{4} = 2 \frac{3}{8} = 2,375.$$

Мы видим, что уже первое приближение к точному значению  $\int_1^2 x^2 dx = \frac{7}{3} = 2,333\dots$  получилось достаточно хорошим и, уж конечно, значительно лучшим, чем в методе прямоугольников.

Удвоим значение  $n$ :  $n = 4$ . Тогда

$$S_2 = \frac{1}{4} \left( \frac{19}{4} + f\left(\frac{5}{4}\right) + f\left(\frac{7}{4}\right) \right) = \frac{1}{4} \left( \frac{19}{4} + \frac{25}{16} + \frac{49}{16} \right) = \\ = \frac{1}{4} \cdot \frac{75}{8} = \frac{75}{32} = 2,343\dots$$

( $19/4$  — значение, полученное в выражении для  $S_1$ ).

При  $n=8$

$$\begin{aligned} S_3 &= \frac{1}{8} \left( \frac{75}{8} + f\left(\frac{9}{8}\right) + f\left(\frac{11}{8}\right) + f\left(\frac{13}{8}\right) + f\left(\frac{15}{8}\right) \right) = \\ &= \frac{1}{8} \left( \frac{75}{8} + \frac{81}{64} + \frac{121}{64} + \frac{169}{64} + \frac{225}{64} \right) = \frac{1}{8} \cdot \frac{1196}{64} = \frac{299}{128} = 2,336... \end{aligned}$$

(75/8 — значение, полученное при вычислении  $S_2$ ).

Мы видим, что  $|S_3 - S_2| \approx 0,007 < 0,01$ , т. е. желаемая точность достигнута, и  $S_3$  можно принять за приближенное значение интеграла

$\int_1^2 f(x) dx$ . Как в примере с методом прямоугольников, так и в примере с методом трапеций мы начинали с  $n=2$ . Разумеется, можно было бы начать и с большего количества точек разбиения.

Для самоконтроля предлагаем читателю вычислить следующие интегралы:

1) Вычислить  $\int_0^1 \frac{dx}{1+x}$  по формуле (28) и формуле (29) при  $n=8$ .

2) Вычислить  $\int_1^3 \frac{dx}{x}$  методом прямоугольников и методом трапеций для  $\varepsilon=0,01$ .

3) Вычислить  $\int_0^{\pi/2} \sin x dx$  методом трапеций для  $\varepsilon=0,01$ . Значения  $\sin x$  брать из таблиц.

## § 2.7. Вычисление некоторых элементарных функций

При решении различных задач часто приходится использовать значения элементарных функций, таких, как  $\sin x$ ,  $\cos x$ ,  $\ln x$ ,  $e^x$  и т. д. Эти значения можно брать из таблиц. Однако соответствующих таблиц может не оказаться под рукой или же нужно знать значение функции в той точке, которой нет в таблице. В последнем случае придется воспользоваться интерполированием. Наконец, нас может не удовлетворить точность таблиц.

Даже если в нашем распоряжении есть микрокалькулятор или ЭВМ, то не исключено, что среди набора операций, которые умеют выполнять эти машины, нет операций, вычисляющих нужные нам элементарные функции. Поэтому для вычисления элементарных функций в математике широко распространено представление этих функций в виде некоторых бесконечных сумм. Мы не будем здесь давать какое бы то ни было математическое обоснование таких пред-

ставлений и ограничимся только тем, что приведем некоторые из них:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots, \quad n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n,$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} \pm \dots,$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots + (-1)^n \frac{x^{2n}}{(2n)!} \pm \dots,$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots + (-1)^{n+1} \frac{x^n}{n} \pm \dots, \quad -1 < x \leq 1.$$

В каждом из указанных разложений точность представления функций будет, вообще говоря, тем выше, чем больше взято слагаемых в сумме. Причем значения самих слагаемых с ростом  $n$  стремятся к нулю. Для функций  $e^x$ ,  $\sin x$  и  $\cos x$  это следует из того, что  $n!$  растёт при увеличении  $n$  быстрее, чем  $x^n$  для любого  $x$ , а для функции  $\ln(1+x)$  это так потому, что значение  $x$  здесь ограничено.

Посмотрим, например, к каким результатам приводит использование формулы для  $e^x$ . Вычислим  $e^{0,5}$ , пользуясь приближенной формулой

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}.$$

Получим

$$e^{0,5} \approx 1 + 0,5 + \frac{(0,5)^2}{2!} + \frac{(0,5)^3}{3!} = 1,6458 \dots$$

Из таблиц имеем

$$e^{0,5} = 1,6487 \dots$$

Совпадение достаточно хорошее.

Для вычисления значений функции с некоторой заданной точностью  $\varepsilon$ , где  $\varepsilon$  — малое положительное число, поступают следующим образом. Вычисляют и суммируют слагаемые до тех пор, пока очередное слагаемое не станет по абсолютной величине меньше  $\varepsilon$ . Полученную сумму (вместе с этим последним слагаемым) и принимают за приближенное значение искомой функции.

Пусть, например, нам нужно вычислить все то же значение  $e^{0,5}$  с точностью  $\varepsilon = 0,001$ . Мы уже получили сумму четырех слагаемых, где последнее было  $(0,5)^3/3! = 0,0208$ , т. е. оно больше, чем  $\varepsilon$ . Добавим еще одно слагаемое  $(0,5)^4/4! = 0,0026$ . Получим  $e^{0,5} \approx 1,6484$ . Поскольку последнее слагаемое больше  $\varepsilon$ , то вычисляем следующее слагаемое:  $(0,5)^5/5! = 0,00026 \dots$  и добавляем его к накопленной ранее сумме:

$$e^{0,5} \approx 1,6487.$$

Последнее слагаемое оказалось меньше, чем  $\varepsilon$ , поэтому мы принимаем вычисленную сумму за приближенное значение  $e^{0,5}$ .

Легко заметить, что каждое слагаемое получается из предыдущего по простой рекуррентной формуле

$$\frac{x^{n+1}}{(n+1)!} = \left( \frac{x^n}{n!} \right) \cdot \frac{x}{n+1}.$$

Поэтому нам не надо было каждое слагаемое вычислять независимо, а можно пользоваться значением предыдущего слагаемого. Например,

$$\frac{(0,5)^5}{5!} = \frac{(0,5)^4}{4!} \cdot \frac{0,5}{5} = 0,0026 \cdot 0,1 = 0,00026.$$

В заключение параграфа предлагаем читателю вычислить значения следующих функций (и сравнить с табличными):

- 1) Вычислить с точностью  $\varepsilon = 0,001$  следующие значения:  
а)  $\sin(\pi/4)$ ;    б)  $\cos(\pi/3)$ .
- 2) Вычислить с точностью  $\varepsilon = 0,01$  значение  $\ln 1,5$ .
- 3) Выписать рекуррентные формулы для слагаемых в разложениях функций:  
а)  $\sin x$ ;    б)  $\cos x$ ;    в)  $\ln(1+x)$ .

## ГЛАВА 3

# ЭВМ — ИСПОЛНИТЕЛЬ АЛГОРИТМОВ

### § 3.1. Структура ЭВМ

Во введении и в предыдущих разделах мы говорили о том, что алгоритм и алгоритмическая система подразумевают существование некоторого исполнителя алгоритма, умеющего выполнять набор конкретных действий. Например, исполнитель может действовать только неразмеченной линейкой и циркулем, или, например, исполнитель — человек, может складывать, вычитать, умножать, делить, читать, писать, запоминать, передвигаться в указанном направлении и т. д. Во введении мы провозгласили тезис о том, что ЭВМ — универсальный исполнитель алгоритмов обработки данных любого содержания, а под данными или информацией мы понимали все то, что может быть представлено в виде строк символов. В этом разделе мы попытаемся рассказать о том, как устроена ЭВМ, на чем основан автоматизм ее работы, о машинном языке для записи алгоритмов.

Прежде чем приступить к содержательному изложению структуры ЭВМ, взаимосвязи ее отдельных модулей, давайте несколько отвлечемся. Возможно, это позволит уяснить в лучшей степени дальнейшее изложение.

Прежде всего заметим, что до появления электронных вычислительных машин не было устройств автоматического запоминания и вывода символьной информации. Можно, конечно, возразить, а как же магнитофонные записи, патефонные пластинки, наконец, книги? Но все перечисленные средства запоминания и воспроизведения, как мы знаем, требуют вмешательства человека, требуют его умения читать, самому выбрать нужную запись. Появление устройств автоматического (т. е. без участия человека) запоминания, автоматической выборки информации для ее последующей автоматической обработки связано с ЭВМ.

Мы еще четко до конца не знаем, как работает мозг человека, в чем заключается механизм сознательной деятельности. Однако с уверенностью можем сказать, что во всех этих проявлениях высшей

нервной деятельности огромная (если не определяющая) роль принадлежит процессам запоминания, выбору нужной для обработки информации и снова запоминания результатов этой обработки.

Появление автоматической памяти у ЭВМ—это важнейший шаг, давший возможность машинам выполнять алгоритмы, которые раньше казались исключительной принадлежностью человеческой деятельности!

Разумеется, машина устроена совсем не так, как человеческий мозг. Наверное и принципы их работы совершенно разные. И тем не менее со дня появления вычислительных машин установилась терминология, «очеловечивающая» ЭВМ. Рассказывая об ЭВМ, мы будем употреблять такие, например, фразы: «Машина запомнила», «Машина выбрала из памяти команду», «Машина расшифровала команду», «Машина закодировала» и т. д.

Современная ЭВМ—это сложная система, состоящая из совокупности механических, электронных и электромеханических устройств.

Наша задача—получить лишь начальное представление об электронных вычислительных машинах и о том, как на них можно работать.

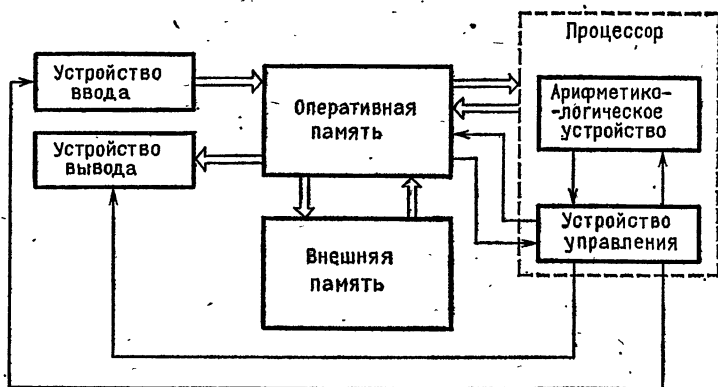


Рис. 3.1

Итак, любая ЭВМ имеет в своем составе следующие устройства:  
— запоминающее устройство (его обычно называют просто памятью);

— процессор, включающий в себя устройство управления и арифметико-логическое устройство (АЛУ);

— устройство ввода;

— устройство вывода.

Эти устройства соединены каналами связи, по которым передается информация. Основные устройства ЭВМ и важнейшие связи между ними представлены на схеме (рис. 3.1). Двойными стрелками показаны



пути и направление движения информации, а простыми стрелками — пути и направление передачи управляющих сигналов.

*Запоминающее устройство (ЗУ)* служит для приема информации из других устройств, ее запоминания, а также выдачи необходимой информации в другие устройства машины.

Память любой ЭВМ обычно состоит из двух частей — оперативной и внешней. В оперативную память можно быстро записать информацию и быстро ее оттуда извлечь.

К оперативной памяти обычно предъявляют два основных требования: быстродействие и большой объем. Они в известной степени технически противоречат друг другу, и потому трудно создавать память машины, которая по обоим этим параметрам удовлетворяла бы пользователей ЭВМ. И хотя в современных ЭВМ оперативная память довольно велика по объему, ее тем не менее не всегда хватает для решения больших задач. Поэтому во всех вычислительных машинах, начиная с машин первого поколения, используют внешнюю память. Мы не будем вдаваться в техническую сущность устройства оперативной и внешней памяти. Скажем только, что внешняя память может быть построена на магнитных лентах (что весьма напоминает обычный магнитофон), магнитных барабанах, дисках. И эта память может быть уже практически сколь угодно большого объема. Зато она и значительно медленнее оперативной, поскольку запись и считывание информации здесь требуют механических перемещений, например движения магнитной ленты вдоль считывающей головки.

Во внешнюю память обычно записывают ту часть информации, которая реже используется в ходе решения задач.

Если снова обратиться к аналогиям, то оперативную память можно сравнить с запоминающей средой нашего мозга, а внешнюю память — с такой внешней запоминающей средой, как записные книжки, справочники, книги, т. е. все то, что отделено от человека, может быть передано другому лицу. Аналогия эта очень близка. Действительно, магнитную ленту, пакет дисков можно «отнять» от машины, унести с собой, положить на склад для хранения, «передать» другой машине.

Оперативную память нельзя «вынуть» из ЭВМ — это ее внутреннее неотъемлемое устройство.

Заметим, что внешняя память может обмениваться информацией лишь с оперативной памятью ЭВМ (это видно и из схемы (рис. 3.1)). Поэтому, чтобы воспользоваться необходимой информацией, хранящейся во внешней памяти, надо сначала вызвать эту информацию в оперативную память. При этом она не пересылается как почтовое отправление, с нее снимается копия, т. е. информация оказывается в оперативной памяти, но она сохраняется и во внешней. Причем информация сохраняется в оперативной памяти до тех пор, пока на это место не будет заслана какая-либо новая информация. Тогда

старая бесследно исчезает. Аналогично происходит и обратная пересылка информации — из оперативной памяти во внешнюю.

Вернемся к оперативной памяти. В память машины вся информация попадает в закодированном виде. Код — это цифровой эквивалент некоторого символа. Как и ранее, мы под информацией будем подразумевать строчки символов, отвлекаясь пока что от их смыслового содержания. Каждому символу (букве, цифре, знаку) можно поставить в соответствие некоторое целое число. Сопоставление символу числового кода называется кодированием. Набор кодов выбирается таким образом, чтобы всегда по закодированной строке можно было бы однозначно восстановить исходную строку.

Например, если мы закодируем буквы а, б, в, г, д, е соответственно числами вида 01, 02, 03, 04, 05, 06, то строка бедабаг примет вид 02060501020104. Такую строку легко расшифровать, зная соответствие букв и цифр. Однако, если мы выберем неудачную кодировку, например такую:

а — 1,  
б — 21,  
в — 11,  
г — 12,  
д — 3,  
е — 13,  
ж — 23,

то та же строка будет закодирована так:

21133121112,

и однозначно расшифровать ее не удастся.

Заметим, что первый и второй случаи кодировки отличаются тем, что в первом каждый символ закодирован строго двумя цифрами, а во втором — либо одной, либо двумя цифрами. В этом случае мы не знаем, какое количество цифр надо выделять на каждый символ в закодированной строке, и соответственно не знаем, какие символы получатся при расшифровке, т. е. мы будем получать разные строки символов, если по-разному интерпретируем набор цифр в закодированной строке. Так, отделив цифры следующим образом:

21 1 3 3 12 11 12,

получим строку символов

баддгвг,

которая, очевидно, не совпадет с первоначально зашифрованной строкой.

Чтобы избежать неоднозначности при кодировке информации, в вычислительных машинах для запоминания информации выделяются строго фиксированные участки памяти. Эти участки называются ячейками памяти. Ячейки все одинаковы и содержат всегда одинаковое число

цифр, составляющих так называемое машинное слово. Это обеспечивает однозначность расшифровки и упрощает технику запоминания и извлечения из оперативной памяти необходимой другим устройствам ЭВМ информации. В каждый разряд ячейки может быть помещена двоичная цифра 0 или 1 (один двоичный разряд называется битом). Когда говорят о длине ячейки или машинного слова, подразумевают число разрядов (битов) в них. Соответствующим образом следует понимать и фразы — «Разрядность машинного слова равна 32 (или 64, или 128, или 8)».

Итак, вся оперативная память состоит из ячеек (слов). Ячейка используется для хранения одного машинного слова, коими могут быть число, команда или некоторое вспомогательное слово (напомним, что в любом случае это последовательность кодов символов фиксированной длины). Объем памяти определяется количеством ячеек. Естественно, чем их больше, тем лучше. В современных ЭВМ объем памяти составляет от нескольких тысяч до нескольких сотен тысяч и даже миллионов ячеек.

Все ячейки равноправны и одинаково доступны для других устройств ЭВМ. Если в какую-либо ячейку записано слово, то оно находится там до тех пор, пока в эту ячейку не будет занесено новое слово. Тогда старое исчезнет. Подчеркнем, что никаких наложений старого и нового слов не происходит, а просто старое стирается и пишется новое. И подобно тому, как это происходит при обмене информацией между оперативной и внешней памятью, при пересылке информации из одной ячейки в другую это слово оказывается в новой ячейке, оставаясь при этом и в ячейке-отправителе.

Любопытно отметить, что в машине Беббиджа предполагалось иметь две разновидности команд пересылки информации из одной ячейки в другую — с сохранением информации в ячейке, из которой она выбирается (т. е. так, как это теперь и делается), и ее затиранием. Все ячейки оперативной памяти последовательно нумеруются, и для того чтобы обратиться к какой-либо ячейке, достаточно указать ее номер. Поэтому по аналогии с почтовым адресом номер ячейки тоже называют *адресом*.

Сейчас принято оценивать объем памяти ЭВМ в байтах. Каждый байт состоит из 8 двоичных разрядов (битов). Содержимое нескольких байтов может составлять одно машинное слово. Байтовая структура памяти позволяет более гибко использовать память ЭВМ, т. е. иметь в ЭВМ длинные и короткие слова, числа, представляемые с различной точностью, и т. д.

Еще несколько слов скажем о внешних запоминающих устройствах. Самые распространенные в настоящее время внешние запоминающие устройства — это магнитные ленты, диски, магнитные барабаны.

Принцип действия этих запоминающих устройств состоит в том, что магнитный носитель информации механически перемещается под

головками магнитной записи или считывания. Поэтому, прежде чем добраться до нужной информации, приходится дожидаться ее появления под головками. В ожидании такого перемещения проходит сравнительно много времени. Например, чтобы добраться до последней записи на магнитной ленте, нужно подождать несколько минут (2—3 мин)— Отсюда возникает понятие времени ожидания, или доступа. Когда начало нужного блока информации оказалось под считывающими головками, то тратится заметное время и на сам процесс считывания.

Перечисленные выше внешние устройства относятся к типу устройств с так называемым последовательным доступом. В них нельзя прямо обратиться к любой требуемой информации, а нужно последовательно до нее добираться. В этом еще одно отличие внешних запоминающих устройств от оперативных.

По принципу действия магнитные диски напоминают грамофонные пластинки. В отличие от магнитной ленты, блок считывающих (записывающих) головок в этом случае перемещается с одной дорожки на другую. На каждой же дорожке сохраняется последовательный принцип считывания (записи). В силу этого время доступа к нужной информации на дисках значительно меньше, чем у магнитных лент. Это время исчисляется сотыми долями секунды.

Магнитные барабаны принадлежат к еще более быстрому запоминающим устройствам. Информация на них записывается также на многих параллельных дорожках. Существенно также и то, что считывание (запись) с этих дорожек может осуществляться параллельно сразу со всех дорожек. Это позволяет сокращать время, идущее на сам процесс считывания (записи) после того, как начало нужного блока информации оказалось под жестко фиксированными головками считывания (записи). Соответственно, на каждую дорожку приходится по своему блоку головок. Но эти затраты окупаются высокой производительностью этих устройств.

В настоящее время стали появляться внешние запоминающие устройства, основанные на использовании опто-электроники. Запись информации производится лазерным лучом и считывание также производится затем с использованием лазера. Эти устройства отличаются высокой плотностью записи, превосходящей плотность магнитной записи в сотни и тысячи раз. Под плотностью записи имеется в виду число двоичных разрядов, которые можно различимо записать на одном погонном миллиметре дорожки. Опто-электронные запоминающие устройства дискового типа в рамках одной дорожки являются последовательными. Сам оптический диск вращается под лазерной считывающей (записывающей) головкой, однако компактность записи уменьшает время доступа и резко сокращает время самого процесса записи (считывания).

Существуют внешние запоминающие устройства, основанные на

принципе магнито-оптики. Лазерным лучом со специально подобранными характеристиками удастся перемагничивать пленку магнитного носителя. Считывание записанной информации также производится с помощью поляризованного лазерного луча. Отражаясь от магнитных участков, луч меняет направление поляризации и тем самым фиксирует наличие или отсутствие бита информации.

Высококачественные опто-электронные внешние запоминающие устройства пока еще сравнительно дороги, но следует ожидать, что в не столь отдаленной перспективе они заменят и магнитные ленты, и магнитные диски, и магнитные барабаны. Применение этой новой техники практически исключит ограничения по объему внешней памяти ЭВМ.

Переработка информации осуществляется в *арифметико-логическом устройстве* ЭВМ. Там выполняются не только арифметические операции, такие, например, как сложение, вычитание, умножение и т. д., но и некоторые логические операции. Кроме таких операций, арифметико-логическое устройство вырабатывает некоторые управляющие сигналы, которые позволяют машине в зависимости от результатов выбирать путь для последующих действий.

В каждой операции участвует специальное запоминающее устройство, называемое регистром сумматора. Информацию для операций, т. е. операнды, арифметико-логическое устройство получает из оперативной памяти ЭВМ. Иногда один из операндов может быть взят непосредственно из сумматора (это зависит от типа ЭВМ). Результат операции может остаться в арифметико-логическом устройстве (сумматоре), либо пересылается в память.

Одним из важнейших узлов электронной вычислительной машины является *устройство управления*. Из самого названия ясно его назначение — управление процессом работы ЭВМ. Оно работает по заданной программе, т. е. последовательности команд, выбирая информацию из запоминающего устройства и из арифметико-логического устройства.

Как уже говорилось, вся информация в ячейках памяти машины записывается в виде некоторой последовательности цифр.

В такой форме в памяти хранятся и команды программы, и числа, и текстовая информация. Возникает вопрос, как же машина различает — число это или команда. Оказывается, по виду последовательности этого сделать нельзя. Все зависит от того, в какое устройство машины она попадает. Если слово передано в арифметико-логическое устройство, то оно рассматривается машиной как операнд (т. е. исходная величина) в какой-либо операции. Если же слово попадает в устройство управления, то оно рассматривается машиной уже как команда.

В этом случае оно дешифруется, т. е. устройство управления по виду команды определяет, какую необходимо выполнить операцию,

над какими операндами, куда поместить результат и откуда взять следующую команду.

Следовательно, очень важно, чтобы слова, содержащие коды чисел, попадали только в арифметико-логическое устройство, а слова, содержащие коды команд,—в устройство управления, но никак не наоборот.

Для того чтобы управлять этим процессом, в устройстве управления есть два регистра—счетчик команд и командный регистр.

Любая последовательность цифр, находящаяся в счетчике команд, рассматривается машиной как номер ячейки, в которой хранится команда, подлежащая исполнению. Например, если в счетчике команд находится слово 1215 (мы пока еще не знаем, как записать такое слово с помощью только нулей и единиц, но уже в следующем параграфе постараемся это объяснить), то слово, хранящееся в ячейке с номером 1215, рассматривается машиной как команда, которую она должна выполнить. Для этого слово из ячейки 1215 поступает в командный регистр, где оно и дешифруется.

Поэтому машинная программа должна быть составлена так, чтобы в счетчик команд поступали лишь номера тех ячеек, в которых содержатся команды. Если же туда попадет номер ячейки, содержащей число, то машина по общему правилу перешлет это число в командный регистр и попытается дешифровать его как команду. В лучшем случае это окажется «глупая» команда, т. е. такая, какой в машине просто не существует, и тогда машина остановится. В худшем случае она выполнит что-то совсем ненужное и, как ни в чем не бывало, продолжит выполнение программы.

*Устройство ввода* служит для ввода в память ЭВМ необходимой информации—программы и исходных данных.

Один из способов ввода информации—ввод с перфокарт, т. е. тот же способ, который предложил еще Бэббидж. Вся информация кодируется специальным образом в виде пробивок на перфокартах. Каждому вводимому символу соответствует определенное расположение пробивок. В устройстве ввода эти комбинации пробивок преобразуются в электрические сигналы, которые далее и передаются в память ЭВМ.

Аналогичным образом осуществляется ввод информации с перфолент. Этот способ очень напоминает телеграфный телетайп.

Оперативная память может заполняться информацией и с внешних запоминающих устройств—магнитных лент, барабанов, дисков.

Ввод информации и управление процессом выполнения программ можно осуществлять и с терминалов. Чаще всего в качестве таких терминалов используются устройства, в состав которых входит телевизионный экран (дисплей) и клавиатура. На экране можно проверить правильность ввода, проследить за процессом решения задачи, в нужный момент внести соответствующие коррективы, посмотреть

на результаты выполнения программы. Причем эти результаты могут быть высвечены в любом удобном для нас виде — графиков, текстов, таблиц и т. д. Тем самым мы уже заговорили о способах *вывода* информации.

Итак, вывести результаты можно на экран, можно на перфокарты и перфоленты, записать на внешние запоминающие устройства, вывести на печать на широкую бумажную ленту алфавитно-цифрового печатающего устройства. В последнем случае результаты могут быть выданы в таком виде, каком нам удобно — можно напечатать тексты, таблицы, рисунки, графики. Для построения графиков машины снабжаются графопостроителями.

### § 3.2. Представление команд в ЭВМ

Любой процесс переработки информации осуществляется с помощью некоторого алгоритма. В ЭВМ такой алгоритм записывается в виде последовательности команд на машинном языке. Причем у каждого типа ЭВМ, вообще говоря, свой машинный язык.

Для каждой конкретной ЭВМ ее конструкцией определен некоторый перечень команд (система команд), которые она способна выполнять. Команда определяет элементарную частичку процесса обработки данных — машинную операцию. Причем у каждой такой операции есть свои исходные данные (операнды) и результат. Результат операции определяется операндами и получается по однозначно определенному для данной операции и конструкции машины правилу.

Выполнение любой машинной операции складывается из следующих действий. В командный регистр устройства управления засылается содержимое ячейки, номер которой содержится в данный момент в счетчике команд (см. § 3.1). Устройство управления рассматривает слово в командном регистре как команду и дешифрует ее. Определяет тип операции, т. е. то, что машина должна сделать. Кроме того выясняются адреса операндов, участвующих в операции. В память поступает запрос на выдачу этих операндов, после чего они поступают в арифметико-логическое устройство. Затем это устройство осуществляет действие над ними по заданной операции и вырабатывает результат, который либо поступает в запоминающее устройство, либо остается в арифметико-логическом устройстве. Наконец, автоматически меняется содержимое счетчика команд, т. е. тем самым определяется, какую команду машина должна будет выполнить следующей. За тем, какую команду надо выполнять следующей, следит устройство управления. Оно, как правило, прибавляет к счетчику команд единицу, что эквивалентно получению адреса следующего машинного слова, которое будет выбрано в качестве очередной команды. Но иногда это общее правило нарушается. Адрес слова, содержащего новую команду, получается не путем прибавления единицы, а ссылкой в счетчик

команд совсем другого адреса. Этот адрес обычно выбирается из предыдущей исполняемой команды, которая называется командой передачи управления. В английском языке для такой команды, резко меняющей содержимое счетчика команд, выбрано образное название — команда скачка (jump instruction).

Итак, команда несет в себе следующую информацию: какую операцию должна выполнить машина, над какими объектами, куда поместить полученный результат.

Как мы знаем, команда может содержать также и информацию о том, какую надо выбрать команду для выполнения в качестве следующей. Каким же образом машина по виду команды определяет всю эту информацию?

Машинное слово, содержащее команду, разбивается на группы разрядов — поля, которые служат для задания информации определенного назначения. Одна группа разрядов отводится под номер операции, или, как говорят, под код операции. Другая группа разрядов — адресное поле — отводится под адреса операндов, участвующих в операции, под адрес результата и, возможно, под адрес следующей по порядку команды. Есть команды, где в адресном поле указывается непосредственно значение одного из операндов (а не адрес операнда, как это бывает обычно).

Основные поля представлены на рис 3.2.

Не следует думать, что в ячейке есть какие-то перегородки, которые отделяют одно поле от другого. В действительности устройство управления, дешифрируя команду, определяет функциональное назначение отдельных ее разрядов, рассматривая определенные совокупности разрядов как поле операций или адресов. Запись команды, например,

Поле операции	Поле адресов
------------------	-----------------

Рис. 3.2

для выполнения арифметического действия в случае, если исходные данные находятся в памяти ЭВМ и результат нужно поместить в память, а адрес следующей команды для выполнения тоже записан в команде, может содержать код операции и четыре адреса: два для операндов, один для указания адреса результата и один для определения местонахождения в памяти следующей по порядку команды.

Машину, у которой команда имеет рассмотренную выше структуру, называют четырехадресной. Команда для нее имеет вид

$\theta A_1 A_2 A_3 A_4$

Здесь  $\theta$  — код операции,  $A_1, A_2, A_3, A_4$  — адреса некоторых ячеек памяти. Например, для арифметических операций:  $A_1$  — адрес первого операнда,  $A_2$  — адрес второго операнда,  $A_3$  — адрес результата,  $A_4$  — адрес следующей команды.

Пусть, например, нужно сложить два числа, находящиеся соответственно в ячейках с номерами 4311 и 3265, результат нужно по-



местить в ячейку 1712. Предположим, что код операции сложения — 01, и пусть команда, выполняющая это действие, расположена в ячейке 1215; а следующая команда, которую надо выполнить после окончания данной команды, находится в ячейке 1216. В этом случае команда в ячейке 1215 должна выглядеть следующим образом:

01 4311 3265 1712 1216

Чтобы это машинное слово было принято машиной к исполнению как команда, число 1215 должно оказаться в счетчике команд (это случилось в результате выполнения предыдущей команды). И, следовательно, слово из ячейки с номером 1215 должно быть заслано в командный регистр устройства управления. Устройство управления по коду операции 01 определяет, что необходимо выполнить сложение. В запоминающее устройство посылаются сигналы на выдачу операндов из ячеек 4311 и 3265. Из памяти содержимое этих ячеек поступает в арифметико-логическое устройство. Там выполняется сложение. Затем это устройство передает результат операции в запоминающее устройство по адресу 1712. После чего в счетчик команд засылается число 1216. Это означает, что именно слово из ячейки 1216 будет воспринято машиной как следующая команда для исполнения. На этом заканчивается выполнение данной команды.

На практике, как правило, нет необходимости указывать в команде все четыре адреса. Составляя программу, обычно подразумевают, что команды выполняются последовательно одна за другой из подряд стоящих по порядку номеров ячеек. В случае, если такой естественный порядок требует изменения, то это производится с помощью специальных команд перехода (скачка). Поэтому довольно широкое распространение, особенно среди ЭВМ первого поколения, получили трехадресные машины:

$\theta A_1 A_2 A_3$

Здесь  $\theta$  — код операции,  $A_1$ ,  $A_2$  — адреса операндов,  $A_3$  — адрес результата.

Если такая команда находится в ячейке с номером  $\alpha$ , то следующая за ней команда будет выполняться из ячейки с номером  $\alpha + 1$ , для чего по окончании выполнения команды к содержимому счетчика команд автоматически добавляется единица. Для того чтобы нарушить эту естественную последовательность операций, нужно воспользоваться специальной командой — командой передачи управления. О ней речь пойдет несколько позже.

Не менее распространены и двухадресные машины. В них команды имеют форму

$\theta A_1 A_2$

В этой системе команд адреса могут иметь различный смысл. Например,  $A_1$  и  $A_2$  — адреса операндов выполняемой операции, результат которой засылается либо по адресу  $A_1$ , либо по адресу  $A_2$ , замеща

ранее выбранный операнд. Возможен также вариант, когда операнды выбираются по адресам  $A_1$  и  $A_2$ , а результат остается в арифметико-логическом устройстве (в регистре сумматора) и служит, как правило, операндом для последующей операции.

Возможен, наконец, случай, когда один из операндов берется из регистра сумматора как результат предыдущей операции, а второй операнд выбирается по адресу  $A_1$  (или  $A_2$ ), результат же помещается по адресу  $A_2$  (или  $A_1$ ).

Вариант интерпретации адресов в ЭВМ с двухадресной системой команд определяется конструкцией машины.

Наконец, одноадресные ЭВМ. В этих машинах в записи команд в явном виде задается лишь один адрес, т. е. команда имеет вид

$\theta A$

При выполнении такой команды один операнд выбирается из оперативной памяти по адресу  $A$ , а вторым операндом служит содержимое сумматора. Результат при этом остается в сумматоре и сохраняется в нем, пока не будет выполнена другая операция, результат которой зашлется в сумматор. Для того чтобы теперь его поместить в память, в системе команд ЭВМ должна быть операция для пересылки содержимого сумматора в память по указанному адресу. Необходимо также предусмотреть и команду для вызова в сумматор содержимого ячейки с указанным в команде адресом.

Пусть, например, нам нужно сложить два числа, хранящиеся в ячейках с адресами  $A_1$  и  $A_2$ , а результат поместить в ячейку с адресом  $A_3$ .

Для решения этой задачи на трехадресной машине, как мы знаем, достаточно выполнить всего одну команду:

$\theta A_1 A_2 A_3$

где  $\theta$  — код операции сложения.

Для решения этой же задачи на одноадресной машине необходимо последовательно выполнить следующие действия:

1) вызвать в сумматор содержимое ячейки  $A_1$  (код операции обозначим  $\theta_1$ );

2) сложить содержимое сумматора с содержимым ячейки  $A_2$  (код операции сложения обозначим  $\theta_2$ ); при этом результат останется в сумматоре;

3) содержимое сумматора переслать в ячейку памяти по адресу  $A_3$  (код операции пересылки  $\theta_3$ ).

То есть для сложения двух чисел потребуется уже не одна команда, как в трехадресной машине, а три:

$\theta_1 A_1$

$\theta_2 A_2$

$\theta_3 A_3$

Однако этот пример вовсе не означает, что для реализации любого алгоритма в одноадресной машине потребуется втрое больше команд, чем в трехадресной. В самом деле, пусть нам надо сложить 100 чисел, хранящихся соответственно в ячейках  $A_1, A_2, \dots, A_{100}$ , и результат поместить в ячейку  $B$ . Программа для трехадресной машины выгляди́т следующим образом ( $\theta$  — код сложения):

$$\begin{array}{l} \theta \ A_1 \ A_2 \ B \\ \theta \ B \ A_3 \ B \\ \theta \ B \ A_4 \ B \\ \dots \dots \dots \\ \theta \ B \ A_{100} \ B \end{array}$$

То есть для сложения ста чисел потребовалось выполнить 99 команд.

Решим теперь задачу на одноадресной машине, приняв те же обозначения, что в примере сложения двух чисел:

$$\begin{array}{l} \theta_1 \ A_1 \\ \theta_2 \ A_2 \\ \theta_2 \ A_3 \\ \dots \dots \dots \\ \theta_2 \ A_{100} \\ \theta_3 \ B \end{array}$$

Здесь потребовалась для сложения ста чисел 101 команда, т. е. всего на две команды больше, чем в трехадресной машине. Все дело в том, что каждая последующая операция здесь использует только что полученный на сумматоре результат и нет необходимости в пересылках из сумматора в память и обратно.

Кроме того, отметим, что время выполнения программы на ЭВМ в значительной степени зависит от числа обращений к памяти. Если теперь вновь обратиться к выполнению операции сложения двух чисел на трехадресной машине, то легко видеть, что мы вынуждены трижды обращаться к памяти, а именно выбрать два аргумента и заслать результат. При выполнении этой операции на одноадресной машине число обращений к памяти остается таким же, как и для трехадресной. Таким образом, время выполнения отдельно взятой операции в обоих случаях остается примерно одинаковым. Если же мы снова рассмотрим пример со сложением ста чисел, то заметим, что общее количество обращений к памяти в программе для трехадресной машины много больше, чем в программе для одноадресной ЭВМ.

Многие современные вычислительные машины конструируются таким образом, что в системе команд можно использовать и одноадресные, и двухадресные, и трехадресные команды. Можно сказать, что такие машины имеют возможность использовать команды разных форматов.

Более того, во многих ЭВМ в адресном поле команды закодиро-

ваны не адреса, а различные правила вычисления адресов. Выполнение команд на этих машинах включает также и действия по вычислению адресов операндов, и в центральном процессоре таких устройств существует своеобразное арифметическое устройство, выполняющее эту работу. В самом деле, во многих алгоритмах операнды, участвующие в арифметических действиях, выбираются один вслед за другим. Естественно в этом случае задать начальный адрес (базовый адрес) этой группы операндов, а в командах указывать только расположение выбираемых операндов относительно этого, зафиксированного в аппаратуре машины базового адреса. Очевидно, чтобы получить адрес операнда, следует к базовому адресу прибавить некоторый добавок — число, указывающее относительное расположение искомого операнда в ячейках памяти. Такой добавок называют смещением, подразумевая смещение относительно некоторой базы. Следовательно, возникает целая процедура, требующая для вычисления адресов выполнения арифметических действий. Различных процедур для получения полного адреса одного или нескольких операндов бывает достаточно много. Известны некоторые типы ЭВМ, в которых используется до 18 различных способов получения адресов операндов!

В дальнейшем изложении мы будем рассматривать только самые простые способы задания адреса операнда — это прямая запись его в полях адреса команды и запись относительно некоторой базы.

Итак, для того чтобы ЭВМ могла начать свою автоматическую работу, необходимо прежде всего разместить в ее оперативной памяти последовательность кодов команд, машине надо указать адрес первой команды, с которой следует начать выполнение программы. Кроме того, в оперативной памяти должны быть размещены коды данных, которые будут использоваться в качестве операндов при выполнении арифметико-логических действий.

Каким же образом эта достаточно сложно закодированная информация о командах и числах оказывается в памяти ЭВМ, кто ее кодирует?

На заре вычислительной техники изготовление цифровых кодов команд и чисел осуществлял человек на бумаге. Затем эта последовательность огромного числа цифр перфорировалась на специальной технике, содержащей только цифровые клавиши. Далее, перфокарты или перфоленты с «пробитыми» командами и числами вводились в ЭВМ с устройств ввода, т. е. в памяти ЭВМ оказывалась копия тех цифр, которые первоначально изготовлял программист.

Программист обязан был точно знать, как кодируются все команды ЭВМ, как кодируются числа и какие-либо другие величины, как все они разместятся после ввода в памяти ЭВМ.

Это было очень трудоемкой и сложной работой.

Современные машины могут начать автоматическую работу, как и прежде, только в том случае, когда в их памяти находится про-

грамма в машинном коде. Однако основную работу по кодировке программ берет на себя сама ЭВМ, более точно — специальные программы.

Но чтобы работали такие программы, они сами должны появиться в памяти ЭВМ в машинно-кодированном виде! Кто же кодирует эти программы, которые в свою очередь будут кодировать другие программы?

На этот вопрос мы постараемся дать ответ в следующих разделах нашей книги.

### § 3.3. Позиционные системы счисления

Под системой счисления понимают совокупность приемов записи и наименования чисел.

Примером системы счисления является всем нам хорошо известная десятичная система счисления. Любое число в ней представляется с помощью набора из десяти цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Причем очень важно, что значение каждой цифры в записи числа зависит от того места, на котором она стоит в этой записи. Так, например, в записи 666,66 цифра 6 встречается пять раз, но в каждой позиции она имеет разный смысл: крайняя левая цифра 6 означает количество сотен, следующая цифра 6 означает количество десятков, 6, стоящая перед запятой, означает количество единиц, цифра 6 после запятой — количество десятых долей единицы и, наконец, последняя цифра 6 — количество сотых долей единицы. Все это можно выразить следующим образом:

$$666,66 = 6 \cdot 10^2 + 6 \cdot 10^1 + 6 \cdot 10^0 + 6 \cdot 10^{-1} + 6 \cdot 10^{-2}$$

Мы видим, что запятая определяет значение каждой позиции в записи числа. Подобным образом можно представить любое десятичное число:

$$\begin{aligned} a_n a_{n-1} a_{n-2} \dots a_1 a_0, a_{-1} a_{-2} \dots = \\ = a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0 + \\ + a_{-1} \cdot 10^{-1} + a_{-2} \cdot 10^{-2} + \dots \end{aligned}$$

Отсюда видно, что цифры  $a_i$  в записи любого числа есть не что иное, как коэффициенты в разложении этого числа по степеням десяти. Каждый символ  $a_i$  есть одна из десятичных цифр. Значение позиции любого символа зависит от расположения запятой в записи числа. Поэтому десятичная система счисления называется позиционной.

В десятичной системе счисления используют десять различных символов — цифр, значение единицы каждого разряда в 10 раз больше единицы соседнего с ним правого разряда. Само число 10 называют *основанием* десятичной системы счисления, а цифры, используемые в десятичной системе, называют *базисными* числами этой системы.

Итак, подчеркнем, что представление чисел в десятичной системе счисления основано на том, что любое число можно разложить по степеням числа 10, где каждый из коэффициентов — одно из базисных чисел этой системы. Последовательность этих коэффициентов и есть запись числа в десятичной системе счисления. Но ведь можно разлагать числа не только по степеням числа 10, а по степеням любого другого целого числа. Разложим, например, число 25,75 по степеням числа 2:

$$25,75 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2}$$

Коэффициенты в разложении здесь представлены одной из двух возможных цифр — 0 или 1. И точно так же, как и в десятичной системе, можно записать число, собрав все коэффициенты при степенях числа 2:

$$11001,11$$

Как и в десятичной системе, запятая отделяет целую часть от дробной и определяет значение каждой позиции в записи числа. Получившаяся здесь запись есть запись числа в двоичной системе счисления: основание системы счисления — число 2, а базисные числа есть 0 и 1.

Чтобы отличать числа, записанные в разных системах счисления, когда это не очевидно из контекста, мы будем заключать их в скобки и внизу писать основание системы счисления:

$$(25,75)_{10} = (11001,11)_2$$

Один из первых создателей электронных вычислительных машин профессор А. Атанасов предложил использовать в ЭВМ именно двоичную систему счисления. Возникает естественный вопрос — чем же эта система лучше десятичной? К десятичной системе мы привыкли, она нам кажется простой и понятной. Оказывается, двоичная система применительно к ЭВМ имеет ряд преимуществ по сравнению с десятичной.

Если нужно представить числа в десятичной системе счисления, то для реализации каждого разряда произвольного числа необходимо иметь 10 различных устойчивых состояний соответствующего запоминающего устройства. В двоичной же системе счисления любое число представляется набором цифр, состоящим только из нулей и единиц, т. е. в каждом разряде нужно уметь представить лишь два устойчивых состояния — одно должно соответствовать нулю, другое — единице. Нет нужды доказывать, что технически реализовать два различных устойчивых состояния проще, чем десять.

Но не только это преимущество двоичной системы привлекло к ней внимание создателей ЭВМ. Главное достоинство двоичной системы счисления — простота ее арифметики. В самом деле, когда мы складываем любые две цифры в десятичной системе, то в сущности

мы пользуемся выученной нами таблицей сложения:  $6+3=9$ ,  $7+9=16$  и т. д. Таблица эта достаточно громоздкая и не так уж легка для запоминания.

Напишем теперь таблицу сложения в двоичной системе счисления:

$$\begin{array}{ll} 0+0=0 & 1+0=1 \\ 0+1=1 & 1+1=(10)_2 \end{array}$$

(напомним, что запись 10 в двоичной системе счисления есть число 2 в десятичной системе, так как  $(10)_2 = 1 \cdot 2^1 + 0 \cdot 2^0 = (2)_{10}$ ).

Еще ярче проявляется простота двоичной арифметики на примере таблицы умножения. Вспомним, сколько усилий пришлось нам затратить в детстве, чтобы выучить таблицу умножения. Вот какова она в двоичной системе счисления:

$$\begin{array}{ll} 0 \times 0 = 0 & 1 \times 0 = 0 \\ 0 \times 1 = 0 & 1 \times 1 = 1 \end{array}$$

Комментарии излишни.

Исключительно просто выполняется в двоичной системе умножение многозначных чисел. Пусть, например, нужно перемножить числа  $(1010)_2$  и  $(101)_2$  (в десятичной системе это соответственно 10 и 5):

$$\begin{array}{r} \times 101 \\ 101 \\ \hline 1010 \\ 1010 \\ \hline 110010 \end{array}$$

Здесь и умножения в привычном смысле слова нет — если соответствующий разряд множителя 1, то множимое надо записать и произвести сдвиг на один разряд; если 0, то только сдвиг на один разряд. А дальше остается, пользуясь таблицей сложения, просуммировать промежуточные результаты. Для проверки правильности умножения выпишем разложение результата по степеням числа 2:

$$(110010)_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

Выписанная сумма дает число  $(50)_{10}$ , т. е. как раз то, что должно было получиться:  $10 \times 5 = 50$ .

Заметим, что и операция деления многозначных чисел в двоичной системе счисления много проще, чем в десятичной.

К недостаткам двоичной системы можно отнести некоторую громоздкость записи чисел по сравнению с десятичной системой. Мы уже убедились, например, что двузначное десятичное число 50 в двоичной записи выглядит как 110010, т. е. для его записи требуется 6 разрядов. Посмотрим, как десятичные цифры записываются в двоич-

ной системе счисления:

$$\begin{aligned}
 0 &= (0)_2 \\
 1 &= 1 \cdot 2^0 = (1)_2 \\
 2 &= 1 \cdot 2^1 + 0 \cdot 2^0 = (10)_2 \\
 3 &= 1 \cdot 2^1 + 1 \cdot 2^0 = (11)_2 \\
 4 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = (100)_2 \\
 5 &= 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (101)_2 \\
 6 &= 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = (110)_2 \\
 7 &= 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = (111)_2 \\
 8 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = (1000)_2 \\
 9 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (1001)_2
 \end{aligned}$$

Безусловно, запись чисел в двоичной системе более громоздка, чем в десятичной. Но указанные выше преимущества двоичной системы с лихвой покрывают этот недостаток.

Кроме двоичной системы счисления нам понадобится в дальнейшем восьмеричная система счисления. Здесь уже основанием системы счисления является 8, а базисными числами 0, 1, 2, 3, 4, 5, 6 и 7. То есть любое число разлагается по степеням числа 8 с указанными выше коэффициентами:

$$\begin{aligned}
 (a_n a_{n-1} \dots a_1 a_0, a_{-1} a_{-2} \dots)_8 = \\
 = a_n \cdot 8^n + a_{n-1} \cdot 8^{n-1} + \dots + a_1 \cdot 8^1 + a_0 \cdot 8^0 + a_{-1} \cdot 8^{-1} + a_{-2} \cdot 8^{-2} + \dots,
 \end{aligned}$$

где  $a_i$  может принимать значение любого из базисных чисел, т. е. от 0 до 7 включительно.

Например,

$$(654,2)_8 = 6 \cdot 8^2 + 5 \cdot 8^1 + 4 \cdot 8^0 + 2 \cdot 8^{-1} = 6 \cdot 64 + 40 + 4 + 0,25 = (428,25)_{10}.$$

Полезно познакомиться и с шестнадцатеричной системой счисления. (Внимательный читатель заметил, что мы рассматриваем системы счисления, где основание является какой-либо степенью числа 2. Разумеется, это не случайно.) Базисные числа здесь от 0 до 15 включительно, т. е. любое число разлагается по степеням числа 16 с коэффициентами из указанного набора базисных чисел. Здесь, однако, возникает проблема обозначения базисных чисел — арабских цифр уже не хватает. Поэтому для обозначения базисных чисел от 0 до 9 используют обычные арабские цифры от 0 до 9, а для последующих чисел — от 10 до 15 — используют в качестве базисных символов либо буквы  $a, b, c, d, e, f$ , либо цифры с черточкой  $\bar{0}, \bar{1}, \bar{2}, \bar{3}, \bar{4}, \bar{5}$ . Тогда запись  $(5e1,4)_{16}$  означает

$$\begin{aligned}
 (5e1,4)_{16} &= (5\bar{4}1,4)_{16} = 5 \cdot 16^2 + 14 \cdot 16^1 + 1 \cdot 16^0 + 4 \cdot 16^{-1} = \\
 &= 5 \cdot 256 + 14 \cdot 16 + 1 + 0,25 = (1505,25)_{10}.
 \end{aligned}$$

Отметим одно важное обстоятельство. Подобно тому, как в десятичной системе счисления при умножении любого числа на  $10^K$  мы сдвигаем запятую на  $K$  разрядов вправо, если  $K$  — положительное число, и  $K$  разрядов влево, если  $K$  — отрицательное, так и в любой



другой системе счисления с основанием  $Q$  при умножении числа на  $Q^K$  мы просто переносим запятую на  $K$  разрядов вправо или влево в зависимости от знака  $K$ .

Например, число  $(13,5)_{10} = (1101,1)_2 = (15,4)_8$  умножим на 8. Так как  $8^1 = 2^3$ , то получим  $(13,5 \cdot 8)_{10} = (108)_{10} = (1101100)_2 = (154)_8$  (перенесли запятую в двоичном изображении числа на три разряда вправо, а в восьмеричном на один вправо).

### § 3.4. Перевод чисел из одной системы счисления в другую

Как бы хороша ни была двоичная система счисления, а все-таки для нас привычнее десятичная система. Поэтому нам удобнее задавать исходные величины для вычислений на ЭВМ и получать результаты в десятичной системе счисления. Значит, нужно уметь переводить числа из одной системы счисления в другую. Сначала займемся переводом чисел из двоичной системы в десятичную.

Мы знаем, что запись любого числа в двоичной системе счисления представляет набор коэффициентов в разложении этого числа по степеням числа 2:

$$x = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2^1 + a_0 \cdot 2^0 + a_{-1} \cdot 2^{-1} + a_{-2} \cdot 2^{-2} + \dots$$

Здесь каждое из  $a_i$  равно либо 0, либо 1.

Для получения десятичного изображения этого выражения достаточно перемножить эти коэффициенты с соответствующими степенями числа 2 и сложить результаты. Все это нужно проделать в десятичной системе счисления.

Например,

$$(1011001,01)_2 = 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 64 + 16 + 8 + 1 + 0,25 = (89,25)_{10}.$$

При переводе целых чисел из двоичной системы счисления в десятичную можно воспользоваться уже нам известным удобным алгоритмом, который носит название *схемы Горнера*. Для этого полином по степеням двойки представляют в следующем виде:

$$\begin{aligned} S &= a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \dots + a_1 \cdot 2 + a_0 = \\ &= (\dots((a_n \cdot 2 + a_{n-1}) \cdot 2 + a_{n-2}) \cdot 2 + \dots + a_1) \cdot 2 + a_0. \end{aligned}$$

Легко видеть, что весь процесс вычисления значения полинома здесь состоит из однотипных операций: умножения предыдущего результата на 2 и последующего сложения его с очередным коэффициентом (на первом шаге на 2 умножается коэффициент  $a_n$ ).

Переведем, например, число  $(1011001)_2$  в десятичную систему счисления:

$$(1011001)_2 = 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 = (89)_{10}.$$

По схеме Горнера имеем

$$(1011001)_2 = (((((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 0) \cdot 2 + 1 = (89)_{10}.$$

Сложнее обстоит дело с переводом чисел из десятичной системы счисления в двоичную.

Здесь нам придется отдельно рассмотреть перевод целых чисел и правильных дробей.

Пусть  $A$  — целое число в десятичной системе счисления. Нам нужно получить его запись в двоичной системе, т. е. она должна иметь вид

$$A = (a_n a_{n-1} a_{n-2} \dots a_1 a_0)_2,$$

где все  $a_i$  нам пока неизвестны, неизвестно и количество этих коэффициентов. Знаем только, что каждое из  $a_i$  равно либо 0, либо 1.

Итак,

$$A = a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \dots + a_1 \cdot 2 + a_0. \quad (2)$$

Разделим обе части равенства (2) на 2. Операцию деления производим в десятичной системе. Причем в левой части мы можем произвести фактическое деление, ибо число  $A$  нам известно, а в правой части деление выполним чисто формально, аналитически:

$$A/2 = a_n \cdot 2^{n-1} + a_{n-1} \cdot 2^{n-2} + a_{n-2} \cdot 2^{n-3} + \dots + a_1 + a_0/2.$$

Нетрудно видеть, что целая часть результата деления будет равна  $a_n \cdot 2^{n-1} + a_{n-1} \cdot 2^{n-2} + \dots + a_1$ , а дробная  $a_0/2$ . Действительно,  $a_0$  равно 0 или 1, следовательно,  $a_0/2 < 1$ .

Запишем это в условных обозначениях:

$$\begin{aligned} [A/2] &= a_n \cdot 2^{n-1} + a_{n-1} \cdot 2^{n-2} + \dots + a_1, \\ \{A/2\} &= a_0/2. \end{aligned}$$

(Запись  $[A/2]$  читается как «целая часть  $A/2$ », запись  $\{A/2\}$  читается как «дробная часть  $A/2$ ».)

Отсюда сразу же находится  $a_0$  — младший коэффициент в разложении (2):

$$a_0 = 2 \cdot \{A/2\}.$$

Эта внешне сложная формула задает остаток от деления числа  $A$  на 2.

Действительно, пусть, например,  $A = 25$ . Тогда  $\frac{25}{2} = 12 \frac{1}{2}$ . Здесь целая часть результата равна 12, а дробная  $1/2$ . После умножения последней на 2 и получим остаток от деления, в данном случае 1.

То есть мы сумели найти младший коэффициент  $a_0$  в разложении произвольного целого числа  $A$  по степеням числа 2. Наша задача теперь — отыскать следующий коэффициент в разложении —  $a_1$ . Но  $a_1$  — это младший коэффициент в разложении целого числа

$$[A/2] = a_n 2^{n-1} + a_{n-1} 2^{n-2} + \dots + a_2 2 + a_1.$$

Поэтому находить его можно точно так же, как мы искали  $a_0$ . Для этого введем обозначение  $A_i = [A/2^i]$  и разделим  $A_i$  на 2:

$$A_i/2 = a_n \cdot 2^{n-2} + a_{n-1} \cdot 2^{n-3} + \dots + a_2 + a_1/2.$$

Теперь уже получим

$$\begin{aligned} [A_i/2] &= a_n \cdot 2^{n-2} + a_{n-1} \cdot 2^{n-3} + \dots + a_2, \\ \{A_i/2\} &= a_1/2. \end{aligned}$$

Тем самым мы нашли следующий коэффициент в разложении (2) —  $a_1$ :

$$a_1 = 2 \cdot \{A_1/2\},$$

т. е.  $a_1$  — остаток от деления целого числа  $A_1$  на 2.

Затем полагаем  $A_2 = [A_1/2]$ . Это снова целое число. Делим его на 2 и т. д. То есть получается последовательность рекуррентных соотношений:

$$\begin{aligned} a_i &= 2 \cdot \{A_i/2\}, \\ A_{i+1} &= [A_i/2], \quad i = 1, 2, 3, \dots \end{aligned} \quad (3)$$

Процесс этот должен когда-нибудь кончиться, так как, как бы ни было велико исходное число  $A$ , мы последовательно делим его на 2, получим, наконец,  $A_{i+1} = 0$ .

Действительно, для любого  $A$  найдется такая степень  $N$ , что будет выполнено  $A/2^N < 1$ , т. е. целая часть результата будет равна 0. Тогда и остаток от деления  $A_{i+1}$  на 2 тоже будет равен 0, т. е. мы найдем последний коэффициент, и он будет старшим коэффициентом в разложении (2). Еще раз заметим, что все операции проделывались в десятичной системе счисления.

Итак, алгоритм нахождения коэффициентов в разложении (2) состоит в последовательном делении исходного числа на 2 и собирании остатков от деления. Подчеркнем, что коэффициенты мы получаем, начиная с младшего и кончая старшим. Поэтому при записи результата в двоичной системе полученные коэффициенты нужно записывать в обратном порядке — сначала старший, потом следующий и т. д.

Вернемся к примеру с переводом числа 25 из десятичной системы в двоичную.

Применяя формулы (3), удобно записать алгоритм перевода в следующем виде:

$$\begin{array}{r} 25 \mid 2 \\ \hline 1 \mid 12 \mid 2 \\ \quad \mid 6 \mid 2 \\ \quad \quad \mid 3 \mid 2 \\ \quad \quad \quad \mid 1 \mid 2 \\ \quad \quad \quad \quad \mid 1 \mid 2 \\ \quad \quad \quad \quad \quad \mid 0 \end{array}$$

Собирая остатки от деления, получим

$$(25)_{10} = (11001)_2$$

(стрелка показывает порядок записи коэффициентов результата).

Итак, мы научились переводить целые числа из десятичной системы счисления в двоичную.

Пусть теперь  $x$  — правильная дробь в десятичной системе счисления. Нам нужно получить ее двоичное представление, т. е.

$$x = a_{-1} \cdot 2^{-1} + a_{-2} \cdot 2^{-2} + a_{-3} \cdot 2^{-3} + \dots \quad (4)$$

Здесь  $a_i$  ( $i = 1, 2, \dots$ ) — неизвестные коэффициенты, принимающие значения либо 0, либо 1, которые и надо найти.

Умножим обе части равенства (4) на 2 (в десятичной системе счисления). В левой части мы это можем сделать фактически, а в правой аналитически:

$$x \cdot 2 = a_{-1} + a_{-2} \cdot 2^{-1} + a_{-3} \cdot 2^{-2} + \dots$$

Легко видеть, что целая часть результата дает старший коэффициент  $a_{-1}$  в разложении (4), а оставшая часть — правильная дробь, т. е.

$$a_{-1} = [x \cdot 2].$$

Если обозначить  $x_1 = a_{-2} \cdot 2^{-1} + a_{-3} \cdot 2^{-2} + \dots$ , то с правильной дробью  $x_1$  можно поступать так же, как и с  $x$ , т. е. умножить  $x_1$  на 2. Тогда

$$x_1 \cdot 2 = a_{-2} + a_{-3} \cdot 2^{-1} + a_{-4} \cdot 2^{-2} + \dots$$

Мы получим  $a_{-2} = [x_1 \cdot 2]$ , т. е.  $a_{-2}$  — целая часть от произведения  $x_1$  на 2.

Дробную часть результата обозначим через  $x_2$ :

$$x_2 = a_{-3} \cdot 2^{-1} + a_{-4} \cdot 2^{-2} + \dots$$

Ее снова умножим на 2 и т. д. Получаем рекуррентные соотношения

$$\begin{aligned} a_{-(i+1)} &= [x_i \cdot 2], \\ x_{i+1} &= \{x_i \cdot 2\}, \quad i = 1, 2, \dots \end{aligned} \quad (5)$$

Процесс продолжается до тех пор, пока не будет получено  $x_{i+1} = 0$ . Причем, в отличие от алгоритма перевода целых чисел, этот процесс может оказаться и бесконечным. Особо подчеркнем, что теперь мы получаем коэффициенты результата в прямом порядке, начиная со старшего.

Рассмотрим два примера.

Пусть надо перевести число  $x = 0,125$  в двоичную систему счисления. Используя рекуррентные формулы (5), можно алгоритм перевода записать в следующем виде (здесь горизонтальная черта отделяет исходное число, а вертикальная черта отделяет получающиеся целые части произведений, которые и являются искомыми коэффициентами в разложении (4)):

$$\begin{array}{r|l} 0 & 125 \\ \hline 0 & 25 \\ 0 & 5 \\ 1 & 0 \end{array}$$

В каждой строке здесь — произведение предыдущей строки на 2. Слева от вертикальной черты — целые части результатов умножения, последовательность которых и дает исходное число в двоичной системе счисления:  $(0,125)_{10} = (0,001)_2$ .

Переведем теперь число  $x = 0,3$ :

0	3
0	6
1	2
0	4
0	8
1	6

Нетрудно видеть, что процесс здесь периодический и никогда не оборвется. До каких же пор надо продолжать вычисления? Если считать, что исходная величина задана точно, то чем больше знаков в результате мы получим, тем лучше. Если это число нужно будет ввести в машину, то достаточно того количества цифр в результате, которое можно записать в ячейку памяти машины. Наконец, если исходное число задано с известной погрешностью, то примерно с той же погрешностью можно брать и переведенное число, т. е. нет нужды получать слишком много знаков в результате.

Итак, мы умеем переводить из десятичной системы счисления в двоичную целые числа и правильные дроби. Так как любое число разлагается на целую и дробную части, то тем самым мы любое число умеем перевести из десятичной системы счисления в двоичную. При этом для перевода целой части мы воспользуемся алгоритмом деления, а дробной части — умножения.

**Пример.** Перевести число 37,25 из десятичной системы счисления в двоичную.

Переводим отдельно числа 37 и 0,25:

$\begin{array}{r l} 37 & 2 \\ \hline 1 & 18 \\ & 0 \\ & 9 \\ & 4 \\ & 2 \\ & 1 \\ & 0 \\ & 1 \\ & 0 \end{array}$	$\begin{array}{r l} 0 & 25 \\ \hline 0 & 5 \\ 1 & 0 \end{array}$
$(37)_{10} = (100101)_2$	$(0,25)_{10} = (0,01)_2$

Таким образом,  $(37,25)_{10} = (100101,01)_2$ . Для того чтобы убедиться в правильности результата, переведем его обратно в десятичную систему счисления:

$$\begin{aligned} (100101,01)_2 &= 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = \\ &= 32 + 4 + 1 + \frac{1}{4} = (37,25)_{10}. \end{aligned}$$

Может возникнуть естественный вопрос — почему при переводе из двоичной системы в десятичную мы пользовались одним алгоритмом, и притом весьма простым, а при переводе из десятичной системы в двоичную — другим, вернее — другими алгоритмами, значительно более сложными? Все дело в том, что в обоих случаях мы пользовались средствами одной и той же, а именно десятичной арифметики. Если бы при переводе из десятичной системы в двоичную мы пользовались двоичной арифметикой, то алгоритм был бы тот же самый, что мы рассматривали при переводе чисел из двоичной системы счисления в десятичную. Приходится все-таки учитывать нашу привязанность к привычной нам десятичной системе счисления.

Мы рассмотрели алгоритм перевода чисел из двоичной системы счисления в десятичную и обратно — из десятичной в двоичную. Алгоритмы останутся совершенно аналогичными, если вместо двоичной системы счисления взять любую другую.

Пусть, например, некоторое число записано в восьмеричной системе счисления. Это означает, что цифры в записи этого числа есть коэффициенты в его разложении по степеням числа 8:

$$(a_n a_{n-1} \dots a_1 a_0, a_{-1} a_{-2} \dots)_8 = a_n \cdot 8^n + a_{n-1} \cdot 8^{n-1} + \dots + a_1 \cdot 8 + a_0 + a_{-1} \cdot 8^{-1} + \dots \quad (6)$$

Для того чтобы получить изображение этого числа в десятичной системе счисления, достаточно выполнить, пользуясь десятичной арифметикой, все операции в правой части выражения (6).

**Пример.** Перевести число  $(276,54)_8$  из восьмеричной системы счисления в десятичную:

$$(276,54)_8 = 2 \cdot 8^2 + 7 \cdot 8^1 + 6 \cdot 8^0 + 5 \cdot 8^{-1} + 4 \cdot 8^{-2} = \\ = 128 + 56 + 6 + \frac{5}{8} + \frac{4}{64} = (190,6875)_{10}.$$

Пусть теперь требуется перевести число из десятичной системы счисления в восьмеричную. Как и в случае перевода в двоичную систему счисления, рассмотрим отдельно целую и дробную части чисел. Для целой части воспользуемся алгоритмом деления, а для дробной — умножения. В первом случае мы получим искомое восьмеричное изображение целого числа, собрав в обратном порядке остатки от последовательного деления на 8, а во втором случае получим восьмеричное изображение дроби, собрав в прямом порядке целые части при последовательном умножении на 8.

**Пример:** Перевести число  $(190,6875)_{10}$  из десятичной системы счисления в восьмеричную.

Переведем целую часть:

$$\begin{array}{r|l} 190 & 8 \\ 16 & 23 \\ \hline 30 & 16 \\ 6 & 7 \end{array} \begin{array}{l} 8 \\ 2 \\ 2 \\ 8 \end{array} \quad (190)_{10} = (276)_8$$

Переведем дробную часть:

$$\begin{array}{r|l} 0 & 6875 \\ \hline 5 & 5000 \\ \hline 4 & 0 \end{array} \quad (0,6875)_{10} = (0,54)_8$$

т. е.  $(190,6875)_{10} = (276,54)_8$ .

Этот пример вместе с предыдущими иллюстрирует, как можно проверять правильность перевода из одной системы счисления в другую обратным переводом.

### § 3.5. Смешанные системы счисления

Мы убедились в том, что перевод чисел из одной системы счисления в другую — занятие довольно трудоемкое. И если ЭВМ работает в двоичной системе, а нам хочется задавать исходные данные и получать результаты в десятичной системе счисления, то нужно проделать большой труд по переводу исходной информации в двоичную систему, а результатов — в десятичную. Тем более что зачастую объем такой информации весьма велик, а точность перевода должна быть очень высокой. Естественно было бы поручить такой перевод самой машине. Но машина воспринимает только последовательности из нулей и единиц. Поэтому нужен простой способ записи десятичных чисел с помощью двоичных цифр. Таким простым способом является представление чисел в смешанной — двоично-десятичной системе счисления. В ней каждая цифра десятичного изображения числа записывается в двоичной системе счисления так, как это делалось в таблице (1). Причем для того, чтобы такая запись была однозначной, для представления любой десятичной цифры отводится одно и то же количество двоичных разрядов, а именно четыре. Если десятичная цифра требует для своего представления меньше значащих двоичных цифр, то впереди этих цифр дописываются нули (так чтобы общее количество двоичных знаков оставалось равным четырем).

Например, десятичное число 834,25 в двоично-десятичной системе запишется так:

$$(834,25)_{10} = (\underbrace{1000}_{(8)} \underbrace{0011}_{(3)} \underbrace{0100}_{(4)}, \underbrace{0010}_{(2)} \underbrace{0101}_{(5)})_{2-10}.$$

Каждая четверка (тетрада) цифр здесь соответствует одной десятичной цифре:

$$\begin{array}{ll} (8)_{10} = (1000)_{2-10} & (2)_{10} = (0010)_{2-10} \\ (3)_{10} = (0011)_{2-10} & (5)_{10} = (0101)_{2-10} \\ (4)_{10} = (0100)_{2-10} & \end{array}$$

Ни в коем случае не следует путать запись числа в двоичной и двоично-десятичной системах счисления. Хотя та и другая используют лишь цифры 0 и 1, но изображение чисел в них разное. Так, если представленное выше двоично-десятичное изображение рассмо-

треть как двоичное, то получится

$$(100000110100,00100101)_2 = 1 \cdot 2^{11} + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^2 + 1 \cdot 2^{-3} + \\ + 1 \cdot 2^{-6} + 1 \cdot 2^{-8} = 2048 + 32 + 16 + 4 + \frac{1}{8} + \frac{1}{64} + \frac{1}{256} = \\ = (2100,14453125)_{10}.$$

Как видим, получившееся десятичное изображение вовсе не совпадает с числом 834,25, из которого мы получили двоично-десятичное изображение.

Можно рассмотреть и более простой пример. Представим число  $(10)_{10}$  в двоичной системе счисления и в двоично-десятичной:

$(10)_{10} = (1010)_2$  (действительно,  $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 2 = 10$ ) и  $(10)_{10} = (00010000)_{2-10}$ , т. е. и в этом случае двоичное и двоично-десятичное изображения одного и того же числа не совпадают. Ну, а есть ли вообще какие-нибудь числа, записи которых в двоичной и в двоично-десятичной системах совпадают?

Из самого определения двоично-десятичной системы счисления следует, что такие числа есть — это цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 десятичной системы счисления. Например,

$$(9)_{10} = (1001)_2 = (1001)_{2-10}.$$

Целая и дробная части записи чисел в двоично-десятичной системе счисления разделяются, как и обычно, запятой. Четверки двоичных цифр отсчитываются налево и направо от запятой. Поэтому при записи целой части двоично-десятичного изображения можно отбрасывать крайние слева нули, а при записи дробной части — крайние справа. Например, десятичное число 524,38 можно записать как таким образом:

$$(0101 \ 0010 \ 0100, \ 0011 \ 1000)_{2-10},$$

так и

$$(101 \ 0010 \ 0100, \ 0011 \ 1)_{2-10}.$$

Обратим внимание на любопытный факт: любой набор двоичных цифр всегда можно рассматривать как двоичное представление некоторого числа, но не любой такой набор может служить двоично-десятичным представлением какого-либо десятичного числа. В самом деле, возьмем, например, следующий набор двоичных цифр:

$$1000010110110010$$

Если его рассматривать как число, записанное в двоичной системе счисления, то получим

$$1 \cdot 2^{15} + 1 \cdot 2^{10} + 1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^1 = \\ = 32768 + 1024 + 256 + 128 + 32 + 16 + 2 = (34226)_{10}.$$

Попробуем теперь рассмотреть исходную двоичную запись как двоично-десятичное представление некоторого десятичного числа.



Разобьем его на четверки цифр

$$\underline{1000} \quad \underline{0101} \quad \underline{1011} \quad \underline{0010}$$

Первая четверка даст цифру 8, вторая — 5, четвертая — 2, а вот третья четверка даст число 11, а вовсе не какую-либо цифру из набора базисных чисел десятичной системы счисления. Следовательно, такая двоичная запись не может являться двоично-десятичным изображением никакого десятичного числа. Подобно рассмотренной нами двоично-десятичной системе можно использовать и другие смешанные системы счисления. Особенно интересны для нас смешанные системы счисления, когда числа, представленные в системе счисления с основанием, равным какой-либо степени числа 2, записываются с помощью двоичных цифр. В первую очередь нас будет интересовать двоично-восьмеричная система счисления, в которой каждая цифра в записи восьмеричного числа представляется в двоичной системе. Для этого достаточно воспользоваться таблицей 1 на с. 95 и для однозначности представления восьмеричного числа в двоично-восьмеричной системе отвести под изображение каждой восьмеричной цифры одно и то же количество двоичных разрядов. Если для двоично-десятичной записи их требовалось для каждой десятичной цифры по 4 разряда, то здесь будет достаточно трех.

Представим, например, восьмеричное число  $(625)_8$  в двоично-восьмеричной записи:

$$(625)_8 = (\underline{110} \quad \underline{010} \quad \underline{101})_{2-8}.$$

Здесь первая тройка двоичных цифр задает число 6, вторая — 2, третья — 5.

Переведем число  $(625)_8$  в десятичную систему счисления:

$$(625)_8 = 6 \cdot 8^2 + 2 \cdot 8^1 + 5 \cdot 8^0 = 384 + 16 + 5 = (405)_{10}.$$

Если мы теперь число  $(405)_{10}$  представим в двоичной системе счисления, т. е. воспользуемся алгоритмом перевода целых чисел из десятичной системы в двоичную:

$$\begin{array}{r} 405 \left| \begin{array}{l} 2 \\ 1 \end{array} \right| \begin{array}{l} 202 \\ 0 \end{array} \left| \begin{array}{l} 2 \\ 1 \end{array} \right| \begin{array}{l} 101 \\ 50 \end{array} \left| \begin{array}{l} 2 \\ 0 \end{array} \right| \begin{array}{l} 25 \\ 1 \end{array} \left| \begin{array}{l} 2 \\ 0 \end{array} \right| \begin{array}{l} 12 \\ 6 \end{array} \left| \begin{array}{l} 2 \\ 0 \end{array} \right| \begin{array}{l} 6 \\ 3 \end{array} \left| \begin{array}{l} 2 \\ 1 \end{array} \right| \begin{array}{l} 3 \\ 1 \end{array} \left| \begin{array}{l} 2 \\ 1 \end{array} \right| \begin{array}{l} 1 \\ 0 \end{array} \end{array}$$

$$(405)_{10} = (110010101)_2,$$

то обнаружим, что двоично-восьмеричная и двоичная записи одного и того же числа  $(625)_8$  совпали. Это могло быть и случайностью, но оказывается справедливой

**Теорема.** Если  $P=Q^n$  ( $P, Q, n$  — целые положительные числа), то запись любого числа в смешанной  $Q-P$ -й системе счисления тождественно совпадает с записью этого же числа в системе счисления с основанием  $Q$  (с точностью до нулей в начале записи целой части числа и в конце дробной).

В частности, если  $P=8, Q=2, n=3$ , то  $8=2^3$  и, следовательно, согласно сформулированной теореме запись любого числа в двоично-восьмеричной системе совпадает с записью этого же числа в двоичной системе. (Заметим, что по той же теореме записи любого числа в двоичной и двоично-шестнадцатеричной системах тоже совпадут.)

Переведем, например, все то же число  $(405)_{10}$  из десятичной системы счисления в шестнадцатеричную:

$$\begin{array}{r|l} 405 & 16 \\ \hline 32 & 25 \\ \hline 85 & 9 \\ \hline 80 & 1 \\ \hline 5 & 10 \end{array}$$

Собирая остатки от деления, получим

$$(405)_{10} = (195)_{16}.$$

Представим теперь число  $(195)_{16}$  в двоично-шестнадцатеричной записи:

$$(195)_{16} = (1 \ 1001 \ 0101)_{2-16}.$$

Нетрудно видеть, что записи одного и того же числа в двоичной и двоично-шестнадцатеричной системах оказались одинаковыми.

Отмеченное свойство двоично-восьмеричной системы счисления позволяет очень просто переводить числа из двоичной системы в восьмеричную (или шестнадцатеричную) и обратно. В самом деле, любую двоичную запись рассматриваем как двоично-восьмеричный код некоторого восьмеричного числа, разбиваем его на тройки (триады) двоичных цифр налево и направо от запятой. Каждой такой тройке ставим в соответствие одну восьмеричную цифру и получаем число в восьмеричной системе счисления. Возьмем, например, код

$$\begin{array}{ccccccc} (10 & 011 & 110 & 001 & 1) & 2 & = (236, 14)_8. \\ \hline 2 & 3 & 6 & 1 & 4 \end{array}$$

Здесь, как и в двоично-десятичной записи, в целой части отброшены крайние слева нули, а в дробной части — крайние справа. Разумеется, надо их учитывать как недостающие в соответствующих триадах двоичных цифр.

Обратный перевод чисел из восьмеричной системы счисления в двоичную столь же прост. Каждую цифру восьмеричного числа

записываем тройкой двоичных символов (пользуемся все той же таблицей 1 на с. 95), т. е. записываем его в двоично-восьмеричной системе, а поскольку эта запись совпадает с двоичной, то мы получим число в двоичной системе счисления.

Переведем, например, число  $(3514,72)_8$  из восьмеричной системы в двоичную:

$$(3514,72)_8 = (\underbrace{11}_3 \underbrace{101}_5 \underbrace{001}_1 \underbrace{100}_4, \underbrace{111}_7 \underbrace{01}_2)_2.$$

Отсюда вытекает, что восьмеричную систему счисления можно использовать для сокращенной записи любого двоичного кода, т. е. для записи любого слова, записанного с помощью нулей и единиц, можно использовать примерно втрое меньше символов, если разбить их на тройки цифр и каждую записать одной восьмеричной цифрой.

Точно так же запись любого числа в шестнадцатеричной системе счисления можно использовать для сокращенной записи любого двоичного кода. В этом случае каждому шестнадцатеричному символу взаимно однозначно соответствует набор из четырех двоичных цифр:

$$\begin{array}{ll} (0)_{16} = (0000)_2 & (8)_{16} = (1000)_2 \\ (1)_{16} = (0001)_2 & (9)_{16} = (1001)_2 \\ (2)_{16} = (0010)_2 & (a)_{16} = (1010)_2 = (10)_{10} \\ (3)_{16} = (0011)_2 & (b)_{16} = (1011)_2 = (11)_{10} \\ (4)_{16} = (0100)_2 & (c)_{16} = (1100)_2 = (12)_{10} \\ (5)_{16} = (0101)_2 & (d)_{16} = (1101)_2 = (13)_{10} \\ (6)_{16} = (0110)_2 & (e)_{16} = (1110)_2 = (14)_{10} \\ (7)_{16} = (0111)_2 & (f)_{16} = (1111)_2 = (15)_{10} \end{array}$$

И так как записи числа в двоично-шестнадцатеричной и двоичной системах по сформулированной выше теореме совпадают, то, заменив все шестнадцатеричные цифры некоторого числа на соответствующие четверки двоичных цифр, получим то же число в двоичной системе счисления. И наоборот, разбив двоичную запись любого числа на четверки (тетрады) цифр влево и вправо от запятой и заменив каждую такую четверку одной шестнадцатеричной цифрой, получим число в шестнадцатеричной системе счисления. При этом запись числа будет использовать примерно в четыре раза меньше цифр, чем в двоичной системе счисления.

Например, число  $(3c2e9)_{16}$  может быть представлено в двоичной системе счисления следующим образом:

$$(\underbrace{11}_3 \underbrace{1100}_c \underbrace{0010}_2 \underbrace{1110}_e \underbrace{1001}_9)_2.$$

(Под каждой четверкой двоичных цифр мы подписали соответствующий этой четверке шестнадцатеричный символ.)

Мы знаем, что любая информация вводится в ЭВМ в виде машинных слов, т. е. последовательности нулей и единиц. Эту информацию, т. е. программу, исходные данные можно записать более коротко, используя восьмеричную или шестнадцатеричную символику.

### § 3.6. Представление чисел в ЭВМ

В современных ЭВМ применяются два способа представления чисел: с фиксированной запятой и плавающей запятой.

В первом случае место запятой, отделяющей целую часть числа от дробной, определяется на этапе конструирования ЭВМ. Сразу же указывается количество разрядов, отводимых для изображения целой и дробной частей. Причем каждому разряду ячейки соответствует всегда один и тот же разряд числа, что существенно упрощает выполнение арифметических действий.

Предположим сначала для простоты, что мы имеем дело с ЭВМ, работающей в десятичной системе счисления, в каждой ячейке памяти которой 10 разрядов и распределены они так: первый разряд отведен для изображения знака числа (знаку «+» соответствует ноль, а знаку «—» соответствует единица), четыре разряда под целую часть, а остальные пять разрядов под дробную часть числа. Тем самым положение запятой, отделяющей целую часть от дробной, строго фиксировано. Тогда самое большое число, которое можно записать в ячейку, есть +9999,99999, самое малое число — 9999,99999. Понятно, что не любое число из этого диапазона может быть точно представлено в ячейке памяти, ведь на дробную часть отведено только пять разрядов. Наименьшее по модулю допустимое число, отличное от нуля, 0000,00001.

Любое неотрицательное число, меньшее чем это, представляется в машине нулем. Если при выполнении каких-либо вычислений получится результат, абсолютная величина которого больше, чем 9999,99999, то старшие разряды полученного числа будут потеряны. Например, если к числу 9999,99998 прибавить число 0000,00005, то мы должны получить 10000,00003. На самом деле в машине результат будет представлен в виде 0000,00003, т. е. совершенно отличный от правильного.

Как преимущества, так и недостатки метода представления чисел с фиксированной запятой очевидны. Основное преимущество — простота арифметических операций, недостаток — слишком узкий диапазон представления чисел.

Все вышесказанное распространяется на ЭВМ, работающие в двоичной системе счисления. Только для получения примерно той же точности представления чисел здесь потребуется примерно втрое большее количество разрядов. Повышение точности в случае изображения чисел с фиксированной запятой возможно лишь за счет увеличения количества разрядов. А это сделать не всегда технически просто.

В различных ЭВМ может быть различная длина ячейки и различные формы представления чисел. Пусть, например, ячейка памяти машины имеет 24 двоичных разряда. Как мы знаем, в ячейку можно

поместить любое машинное слово, т. е. произвольный набор из нулей и единиц. Если это слово—число, то в конструкции машины может быть предусмотрено его представление в форме с фиксированной запятой. В частности, оно может быть таким:

крайний слева разряд—знаковый, затем следующие 9 разрядов отводятся под целую часть и, наконец, оставшиеся 14 разрядов под дробную часть числа, т. е. запятая здесь всегда на одном и том же месте—после десятого разряда машинного слова (с учетом знакового разряда). Тогда самое большое число, представимое в машине, будет

$$(11111111,111111111111)_2.$$

Легко видеть, что оно меньше, чем  $2^9 = (512)_{10}$ . А наименьшее по модулю, допустимое, отличное от нуля число равно

$$(000000000,000000000000001)_2 = 2^{-14}.$$

То есть диапазон чисел, которые можно записать в ячейку памяти машины, здесь таков:

$$2^{-14} \leq |a| < 2^9.$$

Для того чтобы увеличить диапазон представимых чисел, используют другую форму записи чисел—с плавающей запятой. Любое число в системе счисления с основанием  $Q$  можно записать как

$$a = A \cdot Q^P, \quad (7)$$

где  $A$  называют *мантиссой* числа, а  $P$ —*порядком*.

Например, в десятичной системе счисления число 3,14 представим в виде

$$3,14 = 0,314 \cdot 10^1.$$

Здесь мантисса равна 0,314, а порядок 1.

Очевидно, такое представление далеко не однозначно. Можно ведь 3,14 записать так:

$$3,14 = 3,14 \cdot 10^0 = 31,4 \cdot 10^{-1} = 0,0314 \cdot 10^2 = \dots$$

Порядок числа определяет положение запятой в записи мантиссы. При корректировке порядка соответствующим образом меняется и положение запятой—запятая как бы «плавает». Отсюда и название метода представления чисел.

С плавающей запятой число, как мы только что видели, представляется неоднозначно. Одно из этих представлений называют *нормализованным*. В этом случае мантисса должна удовлетворять требованию  $1/10 \leq |A| < 1$  (речь идет о десятичной системе счисления). Иными словами, первая цифра мантиссы после запятой должна быть отличной от нуля. В нашем примере десятичное число  $a = 3,14$  в нормализованной форме имеет вид

$$3,14 = 0,314 \cdot 10^1.$$

Здесь  $A = 0,314$ ,  $P = 1$ . Аналогично, для  $a = -0,00062$  имеем  $-0,00062 = -0,62 \cdot 10^{-3}$ . Здесь  $A = -0,62$ ,  $P = -3$ .

Точно так же в любой другой системе счисления с основанием  $Q$  число  $a$  ( $a \neq 0$ ) записывается в форме с плавающей запятой:  $a = A \cdot Q^P$ , где  $A$  — мантисса,  $P$  — порядок числа  $a$ . Число  $a$  называется нормализованным, если выполняется условие

$$1/Q \leq |A| < 1.$$

В этом случае цифра мантиссы, стоящая после запятой, должна быть отличной от нуля.

В частности, в двоичной системе счисления число  $a$  называется нормализованным, если оно определяется в виде

$$a = A \cdot 2^P,$$

где  $P$  — двоичный порядок,  $A$  — двоичная мантисса числа  $a$ , причем

$$0,5 \leq |A| < 1.$$

Рассмотрим несколько примеров представления чисел в двоичной нормализованной форме.

Пусть задано число  $a = (6)_{10}$ . В двоичной системе счисления оно имеет вид  $(110)_2$ , а в двоичной нормализованной форме  $a = 0,11 \cdot 2^3$ , или, используя только двоичные цифры, получим  $a = 0,11 \cdot 10^{11}$ .

(Напомним, что  $(2)_{10} = (10)_2$ ,  $(3)_{10} = (11)_2$ , т. е.

$$A = (0,11)_2, P = (11)_2.)$$

Для  $a = 3/32$  нормализованное двоичное представление имеет вид

$$a = (3/32)_{10} = 0,11 \cdot 10^{-11},$$

т. е.  $A = (0,11)_2$ ,  $P = -(11)_2$ .

Рассмотренный выше способ записи чисел справедлив для чисел, отличных от нуля. Если же  $a = 0$ , то условимся считать, что в записи  $a = A \cdot Q^P$  в этом случае  $A = 0$  и  $P = 0$ .

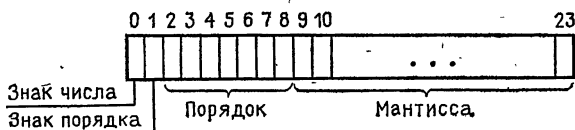


Рис. 3.3

При представлении чисел с плавающей запятой в ячейке памяти ЭВМ выделяют группы разрядов для изображения мантиссы, порядка, знака числа и знака порядка. Если в ячейке 24 разряда, то, пере-  
 нумеровав их с нулевого номера по двадцать третий, можно распре-  
 делить их, например, следующим образом: нулевой разряд отвести  
 под знак числа, первый — под знак порядка, в следующих семи раз-  
 рядах, т. е. со 2-го по 8-й — порядок, и, наконец, с 9-го по 23-й раз-  
 ряды отводятся под мантиссу числа. Причем знак «+» обозначается 0,  
 а «-» обозначается единицей как для знака числа, так и для знака  
 порядка (можно принять и другое соглашение, что, впрочем, и делается  
 в некоторых ЭВМ) (рис. 3.3).

Покажем на примерах, как записываются некоторые числа в ячейке памяти ЭВМ. Пусть  $a = (4)_{10} = (100)_2 = 0,1 \cdot 10^{11}$ . Здесь  $A = (0,1)_2$ ,  $P = (11)_2$ . В ячейке это будет выглядеть следующим образом (рис. 3.4):

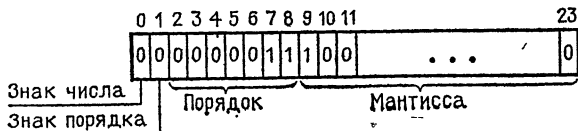


Рис. 3.4

число  $a = -(9,5)_{10} = -(1001,1)_2 = -(0,10011 \cdot 10^{100})_2$  в 24-разрядной ячейке будет представлено в виде (рис. 3.5):

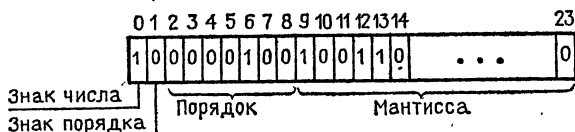


Рис. 3.5

Число «нуль» в форме с плавающей запятой изображается так: (рис. 3.6):

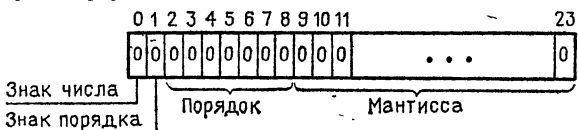


Рис. 3.6

Мы уже говорили ранее, что при записи числа в нормализованном виде мантисса не имеет права начинаться с нуля. Это значит, что такая форма записи дает максимально возможную точность при данном фиксированном количестве разрядов, отведенных под мантиссу числа. При этом обеспечивается и достаточно широкий диапазон чисел, представимых в машине. Минимальное число, которое можно записать в ячейку, есть (рис. 3.7):

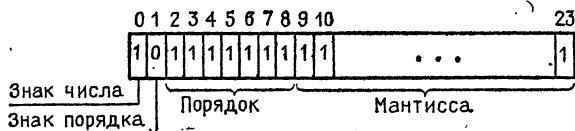


Рис. 3.7

т. е. порядок  $P = (1111111)_2 = (127)_{10}$ , а мантисса  $A = (0, \underbrace{111...}_{15})_2$

весьма близка к единице.

Такое же по абсолютной величине, но положительное число, максимальное из возможно представимых в памяти ЭВМ (рис. 3.8):



Рис. 3.8

Минимальное по модулю, отличное от нуля и нормализованное число  $a = (0,1 \cdot 10^{-111111})_2 = \frac{1}{2} 2^{-127} = 2^{-128}$  (рис. 3.9):

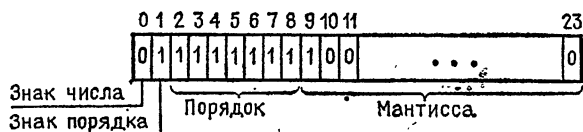


Рис. 3.9

Заметим, что наименьшее по модулю число, не равное нулю и не нормализованное, представимое в ячейке, есть

$$a = \left(\frac{1}{2}\right)^{+15} 2^{-127} = 2^{-142}.$$

В этом случае мантисса

$$A = (0, \underbrace{000 \dots 01}_{15})_2 = 2^{-15}, \text{ порядок } P = -(1111111)_2 = -(127)_{10}.$$

На точность представления чисел влияет не только порядок, но и количество разрядов, отводимых под мантиссу числа.

Например, число  $\left(\frac{1}{3}\right)_{10}$  изображается бесконечной двоичной дробью  $(0,01010101\dots)_2$ , или в нормализованной форме

$$(1/3)_{10} = 0,10101\dots \cdot 2^{-1} = (0,10101\dots 10^{-1})_2$$

$$(A = 0,10101\dots; P = -1).$$

Так как под изображение цифр мантиссы в ячейке памяти ЭВМ отводится ограниченное число разрядов (в нашем случае 15), то учитываются только старшие разряды, а младшие отбрасываются. И число  $(1/3)_{10}$  запишется в ячейке следующим образом:

$$\underbrace{01\ 0000001}_{\text{порядок}} \underbrace{101010101010101}_{\text{мантисса}}$$

Чтобы повысить точность представления чисел, можно увеличить количество разрядов ячейки, отводимых под изображение мантиссы.

Сделать это в пределах одной ячейки можно лишь за счет сокращения числа разрядов, отводимых для изображения порядка. Тем самым мы сузим диапазон чисел, с которыми может работать наша ЭВМ.



Довольно часто для повышения точности представления чисел используют запись чисел с так называемой двойной точностью. В этом случае число записывается не в одной, а в двух подряд идущих ячейках памяти, причем вторая ячейка используется для записи последующих цифр мантиссы. В этом случае для выполнения операций над числами с двоичной точностью нужны специальные команды.

Итак, мы убедились в том, что форма представления чисел с плавающей запятой позволяет записывать числа из весьма широкого диапазона и с достаточно высокой точностью. Это, безусловно, дает такой форме серьезные преимущества перед способом записи чисел с фиксированной запятой. Но есть у нее и существенный недостаток — значительное усложнение арифметических операций.

Пусть, например, нужно сложить два числа:  $(15,5)_{10}$  и  $(0,125)_{10}$ . Прделаем это отдельно — для чисел, представленных с фиксированной запятой и с плавающей запятой, причем как в десятичной, так и в двоичной системе счисления.

Если числа записаны в форме с фиксированной запятой, то целые и дробные части слагаемых находятся на соответствующих местах. И сложение идет обычным образом:

$$\begin{array}{r} + 15,5 \\ 0,125 \\ \hline 15,625 \end{array}$$

Иное дело, если слагаемые записаны в форме с плавающей запятой. Тогда  $15,5 = 0,155 \cdot 10^2$ , а  $0,125 = 0,125 \cdot 10^0$ . Здесь уже для получения результата необходимо предварительно выравнивать порядки слагаемых. Для этого второе слагаемое запишем в виде  $0,125 = 0,00125 \cdot 10^2$ . Тем самым оно перестало быть нормализованным, зато порядки слагаемых теперь одинаковы, и, значит, мантиссы можно складывать:

$$\begin{array}{r} + 0,155 \\ 0,00125 \\ \hline 0,15625 \end{array}$$

В итоге получим, что мантисса результата равна  $A = 0,15625$ , а порядок  $P = 2$ , т. е. искомая сумма запишется в виде  $0,15625 \cdot 10^2$ . Могло случиться, что результат сложения оказался бы ненормализованным (т. е. мантисса вышла бы за пределы полуинтервала  $[0,5; 1)$ ). В этом случае потребовалось бы еще нормализовать результат, т. е. сдвинуть мантиссу на соответствующее количество разрядов и на столько же изменить порядок результата (при сдвиге мантиссы вправо порядок придется увеличить, влево — уменьшить).

Прделаем ту же процедуру сложения чисел в двоичной системе счисления:

$$\begin{aligned} (15,5)_{10} &= (1111,1)_2, \\ (0,125)_{10} &= (0,001)_2. \end{aligned}$$

В случае представления чисел в форме с фиксированной запятой сложение выполняется элементарно:

$$\begin{array}{r} + 1111,1 \\ 0,001 \\ \hline 1111,101 \end{array}$$

Пусть теперь слагаемые записаны в форме с плавающей запятой и нормализованы. Тогда  $1111,1 = 0,11111 \cdot 10^{100}$ ;  $0,001 = 0,1 \cdot 10^{-10}$

$$((10)_2 = (2)_{10}; (100)_2 = (4)_{10}).$$

Выравниваем порядки:

$$0,1 \cdot 10^{-10} = 0,0000001 \cdot 10^{100}.$$

Теперь сложим мантиссы слагаемых:

$$\begin{array}{r} + 0,11111 \\ 0,0000001 \\ \hline 0,1111101 \end{array}$$

Получим

$$A = 0,1111101, P = 100,$$

т. е. результат сложения равен

$$0,1111101 \cdot 10^{100} = 1111,101,$$

т. е. то же число, что получено при сложении с фиксированной запятой.

Все сказанное выше относится и к операции вычитания, где прежде, чем вычесть одно число из другого, необходимо выравнивать порядки, а после вычитания мантиссы нормализовать результат (если он получился ненормализованным). При операциях умножения и деления не требуется выравнивать порядки. При умножении двух чисел их мантиссы перемножаются, порядки складываются, после чего производится нормализация результата.

Перемножим, например, два числа:  $(20)_{10}$  и  $(0,5)_{10}$ . В десятичной системе счисления при использовании формы представления с плавающей запятой получим

$$(0,2 \cdot 10^2) \cdot (0,5 \cdot 10^0) = 0,1 \cdot 10^2 = 10.$$

В двоичной системе счисления эти сомножители соответственно равны

$$\begin{aligned} (20)_{10} &= (10100)_2 = (0,10100 \cdot 10^{101})_2 = (0,101 \cdot 10^{101})_2, \\ (0,5)_{10} &= (0,1)_2 = (0,1 \cdot 10^0)_2. \end{aligned}$$

Поэтому, перемножив эти числа, получим

$$(0,101 \cdot 10^{101})_2 \cdot (0,1 \cdot 10^0)_2 = (0,0101 \cdot 10^{101})_2 = (0,101 \cdot 10^{100})_2 = (1010)_2 = (10)_{10}.$$

Аналогично поступают при делении двух чисел. Здесь уже делят мантиссу делимого на мантиссу делителя, а из порядка делимого вычитают порядок делителя. Затем частное нормализуют, если оно не нормализовано.

Возьмем те же числа  $(20)_{10}$  и  $(0,5)_{10}$  и поделим одно на другое в десятичной и двоичной системах счисления:

$$(0,2 \cdot 10^2)_{10} : (0,5 \cdot 10^0)_{10} = \frac{0,2}{0,5} \cdot 10^2 = (0,4 \cdot 10^2)_{10} = (40)_{10}$$

(здесь результат сразу же получился нормализованным)

$$(0,101 \cdot 10^{101})_2 : (0,1 \cdot 10^0)_2 = (1,01 \cdot 10^{101})_2 = (0,101 \cdot 10^{119})_2 = (101000)_2 = (40)_{10}$$

(напомним, что  $(110)_2 = (6)_{10}$ ).

## ГЛАВА 4

### ПРИЕМЫ ПРОГРАММИРОВАНИЯ

#### § 4.1. Некоторые способы записи алгоритмов

Любая мысль становится мыслью лишь тогда, когда она сформулирована; любой алгоритм становится алгоритмом лишь тогда, когда он обретает какую-либо форму, неважно, — словесное ли это описание, представление в виде последовательности формул, или в виде программы на языке машины, или еще каким бы то ни было образом. И как для мысли важна словесная оболочка, в которую она облечена, так и для алгоритма исключительно важна форма его изложения. Эта форма существенным образом зависит от той цели, которую мы преследуем при записи алгоритма. Одно дело, когда вновь созданный алгоритм нужно опубликовать, чтобы он стал всеобщим достоянием. В этом случае главное в форме записи — ее наглядность. Алгоритм должен быть легко понят и понят только так, как его трактует автор. Другое дело, когда алгоритм пишется для его непосредственной реализации на ЭВМ. В первом случае можно широко использовать общепринятую математическую символику, многое пояснить словами и т. д. Во втором же случае нужна исключительно строгая формализация, быть может, в ущерб наглядности, зато в такой форме алгоритм может однозначно понять машина.

Мы здесь рассмотрим некоторые наиболее простые формы записи алгоритмов. Естественно, при этом будем стремиться к тому, чтобы запись любого алгоритма была понятна каждому исполнителю, и каждый исполнитель выполнял бы алгоритм одинаково, т. е. для одних и тех же исходных данных в процессе его работы получались бы одинаковыми как все промежуточные, так и окончательные результаты.

Пусть, например, нужно записать алгоритм исследования и решения линейного уравнения  $ax=b$ . Исследовать — значит ответить на вопросы — существует ли решение, в каком случае решение одно, в каком случае решений много. Напомним, что если  $a \neq 0$ , то существует единственное решение  $x=b/a$ ; если  $a=0$  и  $b=0$ , то любое значение  $x$  является решением; если  $a=0$  при  $b \neq 0$ , то уравнение

не имеет решений. В сущности, это уже и есть описание алгоритма исследования и решения уравнения  $ax=b$ .

Этот же алгоритм можно записать несколько иначе. Все действия разобьем на отдельные пункты, после чего алгоритм примет вид:

1. Если  $a \neq 0$ , то перейти к п. 2, в противном случае к п. 4.
2. Вычислить  $x=b/a$  и отпечатать значение переменной  $x$ .
3. Перейти к п. 8.
4. Если  $b \neq 0$ , то перейти к п. 5, в противном случае к п. 7.
5. Отпечатать текст: «Решений нет».
- 6. Перейти к п. 8.
7. Отпечатать текст: «Любое значение  $x$  является решением».
8. Конец алгоритма (Стоп).

Здесь в п. 1 проверяется условие равенства нулю коэффициента  $a$ . Если  $a \neq 0$ , то вслед за п. 1 должен выполняться п. 2, если же  $a=0$ , то необходимо выполнять п. 4. Точно такой же смысл у п. 4. Здесь проверяется значение коэффициента  $b$ , и в случае, если  $b \neq 0$ , происходит переход к выполнению п. 5, а при  $b=0$  — п. 7.

Заметим, что после п. 2 должен выполняться п. 3 алгоритма, после п. 5 — п. 6, а после п. 7 — п. 8. И вообще, если в каком-либо пункте алгоритма нет проверки условия и нет специального указания на то, какой пункт должен выполняться следующим, то по окончании действия, указанного в этом пункте, происходит переход к следующему по номеру пункту.

В этом же алгоритме есть пункты, единственное назначение которых — нарушить естественный порядок выполнения пунктов и осуществить переход не к следующему по порядку, а к тому, который специально указан в данном пункте. Таковыми здесь являются п. 3 и 6. Начинаящие программисты часто забывают про такие переходы, а они в нашем алгоритме совершенно необходимы. Действительно, если мы забудем, например, поставить п. 3, то может сложиться следующая ситуация. Пусть  $b \neq 0$  и  $a \neq 0$ , тогда выполняются пп. 1 и 2, т. е. вычисляется  $x=b/a$ , и так как по нашему предположению п. 3 в алгоритме отсутствует, то вслед за п. 2 выполнится п. 4. В нашем случае  $b \neq 0$ , следовательно, произойдет переход к п. 5 и напечатается текст: «Решений нет», что, конечно же, неверно. Для того-то нам и нужен п. 3, чтобы в случае  $b \neq 0$  и  $a \neq 0$  после вычисления  $x=b/a$  обойти выполнение всех остальных пунктов. Аналогичную роль играет п. 6. Представим себе, что его нет. Тогда в случае  $b \neq 0$  и  $a=0$  выполняются последовательно пункты: 1, 4, 5, 7 и 8. То есть печатается текст «Решений нет» и это правильно, но, к сожалению, на этом алгоритм не заканчивает свою работу (после п. 5 выполняется п. 7) и печатается текст: «Любое значение  $x$  является решением», а это уже неверно.

Запись п. 1 можно несколько упростить, а именно:

1. Если  $a=0$ , то перейти к п. 4.

Здесь подразумевается, что если  $a \neq 0$ , то произойдет естественный переход к следующему по порядку пункту, т. е. к п. 2. Аналогично можно упростить и запись п. 4:

4. Если  $b = 0$ , то перейти к п. 7.

Здесь при  $b \neq 0$  после п. 4 выполняется п. 5.

Рассмотрим еще один пример. Пусть даны числа  $x_1, x_2, x_3$ . Найти число  $u$ , равное наибольшему из них. Решение задачи можно получить, действуя следующим образом. Вначале найдем наибольшее из двух чисел, например,  $x_1$  и  $x_2$ . Если  $x_1$  меньше, чем  $x_2$ , то положим  $u = x_2$ ; если же  $x_1$  не меньше, чем  $x_2$ , то положим  $u = x_1$ . Таким образом, в результате сравнения  $x_1$  с  $x_2$  мы получим величину  $u$ , которая равна наибольшему из них. Сравним теперь  $u$  с  $x_3$ . Если  $u \geq x_3$ , то значение  $u$  следует принять в качестве результата. Если же  $u < x_3$ , то  $u$  следует положить равным  $x_3$ . Таким образом, в качестве результата получим величину  $u$ , которая равна наибольшему из  $x_1, x_2, x_3$ .

Алгоритм решения данной задачи можно представить более четко следующим образом:

1. Если  $x_1 \geq x_2$ , то перейти к п. 4.

2.  $u$  положить равным  $x_2$ .

3. Перейти к п. 5.

4.  $u$  положить равным  $x_1$ .

5. Если  $u \geq x_3$ , то перейти к п. 7.

6.  $u$  положить равным  $x_3$ .

7. Стоп.

Здесь в пп. 1—4 происходит выбор наибольшего из чисел  $x_1$  и  $x_2$  и оно принимается в качестве  $u$ , а в пп. 5, 6 уже это  $u$  сравнивается с  $x_3$  и наибольшее из них становится решением задачи. Как и в предыдущем алгоритме, здесь есть три группы операций. В одной из них осуществляются сравнения различных величин и выбор следующего пункта для исполнения в зависимости от результата сравнения. В другой — указываются переходы на какие-либо пункты алгоритма. В третьей — все остальные. Например, к первой группе относятся пп. 1, 5, ко второй — п. 3; к третьей — пп. 2, 4, 6, 7.

Мы еще несколько усовершенствуем средства записи алгоритмов, если применим часто используемое обозначение — вместо слов «положить равным» будем писать сочетание символов  $:=$ , как это принято во многих реальных языках программирования. Тогда пп. 2, 4 и в последнем алгоритме будут выглядеть следующим образом:

2.  $u := x_2$ .

• • • •

4.  $u := x_1$ .

• • • •

6.  $u := x_3$ .

• • • •

Особо обратим внимание на запись вида  $x := x + A$ . Она означает, что к текущему значению переменной  $x$  прибавляется  $A$ , и этот результат

становится новым значением переменной  $x$ . (Произносят эту запись так: « $x$  присвоить значение  $x + A$ ».)

Все рассмотренные выше примеры алгоритмов имеют так называемый линейный характер, т. е. каждый пункт вычислительного процесса выполняется не более одного раза. При этом некоторые из них, при определенном наборе исходных данных, могут вообще не выполняться ни разу. Так, в случае исследования уравнения  $ax = b$  при  $a \neq 0$  для решения задачи достаточно выполнить пп. 1, 2, 3, 8. Пункты 4—7 при этом не выполняются вообще.

Рассмотрим теперь пример так называемого *циклического процесса*. Пусть надо вычислить сумму элементов последовательности

$$S = \sum_{i=1}^n x_i = x_1 + x_2 + \dots + x_n.$$

Если зафиксируем число  $n$ , то все этапы вычислений можно записать в явном виде. Например, для  $n=6$  получим:

1.  $S := x_1$ .
2.  $S := S + x_2$ .
3.  $S := S + x_3$ .
4.  $S := S + x_4$ .
5.  $S := S + x_5$ .
6.  $S := S + x_6$ .
7. Стоп.

Здесь все действия выполняются последовательно, одно за другим и не более чем по одному разу, т. е. алгоритм, записанный в таком виде, носит линейный характер. В п. 1 переменной  $S$  присваивается значение  $x_1$  и к моменту выполнения п. 2 значение  $S$  равно  $x_1$ . Поэтому в п. 2, где  $S := S + x_2$ , текущее значение  $S$ , которое к этому моменту равно  $x_1$ , складывается с  $x_2$ , и этот результат становится новым значением  $S$ , т. е. значение  $S$  равно сумме  $x_1$  и  $x_2$ . Это новое значение  $S$  в п. 3 алгоритма складывается с  $x_3$ , после чего  $S$  становится равным сумме трех величин  $x_1$ ,  $x_2$  и  $x_3$  и т. д. В п. 6 уже накопленная сумма

$\sum_{i=1}^5 x_i$  складывается с  $x_6$  и  $S$  присваивается этот последний результат.

Легко видеть, что в данном вычислительном процессе на шагах с номерами 2, 3, 4, 5, 6 выполняются, по существу, одинаковые действия. При этом каждый раз переменной  $S$  присваивается значение, равное сумме текущего значения  $S$  и значения очередного слагаемого  $x_i$ , которое определяется значением индекса  $i$  ( $i$  меняется от 1 до 6). Учитывая это обстоятельство, запишем алгоритм следующим образом:

1.  $i := 2$ .
2.  $S := x_1$ .
3.  $S := S + x_i$ .
4.  $i := i + 1$ .

5. Если  $i \leq 6$ , то перейти к п. 3.

6. Стоп.

В п. 1 индексу  $i$  присваивается начальное значение 2. В п. 2 искомой сумме  $S$  присваивается начальное значение — значение переменной  $x_i$ . Эти пункты и п. 6. выполняются по одному разу. А вот пп. 3, 4 и 5 выполняются многократно, — в данном случае 5 раз. Первый раз  $S$  присваивается значение  $S + x_2$ . Затем  $i$  увеличивается на 1, т. е.  $i$  становится равным 3. В п. 5 это значение сравнивается с 6, и, так как 3 меньше, чем 6, то происходит переход к п. 3. Здесь теперь фактически выполняется действие  $S := S + x_3$ . Затем  $i$  снова увеличивается на 1, снова сравнивается с 6 и т. д., — до тех пор, пока в последний раз  $S$  не присвоится значение суммы  $S + x_i$  ( $i$  в этом случае будет равно 6). В очередной раз  $i$  увеличивается на 1, становится равным 7, и в этом случае после п. 5 выполнится уже не п. 3, а п. 6, так как условие  $i \leq 6$  окажется не выполненным и алгоритм закончит свою работу. При этом в  $S$  будет накоплена искомая сумма.

Еще раз подчеркнем, что пп. 3—5 записаны в алгоритме один раз, а выполняются многократно, т. е. образуют так называемый цикл. Особенно важно, что количество пунктов в цикле совершенно не зависит от количества повторений этих действий, т. е. если бы нам надо

было вычислить  $S = \sum_{i=1}^{1000} x_i$ , то запись алгоритма осталась бы неизменной, за исключением п. 5 — здесь надо было бы значение  $i$  сравнивать не с числом 6, а с числом 1000:

5. Если  $i \leq 1000$ , то перейти к п. 3.

В этом случае пп. 3—5 выполнялись бы уже 1000—1 раз.

Аналогичным образом можно записать алгоритм определения наибольшего из  $n$  чисел  $x_1, x_2, \dots, x_n$  ( $n \geq 2$ ), т. е.

$$u = \max_{1 \leq i \leq n} \{x_i\}.$$

Он будет иметь вид:

1.  $i := 1$ .
2.  $u := x_i$ .
3.  $i := i + 1$ .
4. Если  $u \geq x_i$ , то перейти к п. 6.
5.  $u := x_i$ .
6. Если  $i < n$ , то перейти к п. 3.
7. Стоп.

Здесь пп. 1, 2 и 7 выполняются по одному разу, а пп. 3—6 образуют цикл, выполняющийся  $n-1$  раз.

З а м е ч а н и е. Как в рассмотренном алгоритме, так и в последующих алгоритмах, использующих циклы, последовательность дейст-



вий, образующих цикл, для наглядности выделяется нами квадратной скобкой.

Отметим, что надо быть внимательным при записи пункта, в котором проверяется условие на завершение циклического процесса. Так, в примере с вычислением суммы шести чисел в п. 5 такое условие содержало нестрогое неравенство  $i \leq 6$ , а в примере с определением наибольшего из  $n$  чисел условие в п. 6 записывалось с помощью строгого неравенства  $i < n$ . Различие это вызвано тем, что в первом примере значение индекса  $i$  менялось после того, как использовалась переменная с этим индексом, а во втором наоборот, — сначала изменялось значение индекса  $i$ , а затем уже в цикле использовалась переменная с этим индексом. Разумеется, строгое неравенство можно было бы использовать и в первом примере, но тогда оно выглядело бы следующим образом:  $i < 7$ . Обычно для того, чтобы убедиться в правильности записи условия, проверяют работу алгоритма для конкретных значений величин, определяющих число повторений циклического процесса. Например, для проверки алгоритма выбора наибольшего из  $n$  чисел полагают  $n$  равным двум или трем и, как говорят, «прокручивают» алгоритм при этом конкретном значении  $n$ .

Как мы знаем, один и тот же результат можно получить, если пользоваться разными, но эквивалентными алгоритмами. Поэтому из нескольких таких алгоритмов выбирают один, представляющийся наиболее удобным, более наглядным, требующий меньшего количества операций.

Пусть, например, надо вычислить значение многочлена  $P_n(x)$  степени  $n$  в некоторой точке  $x$ . Многочлен можно представить в традиционной форме

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0,$$

а можно и в соответствии со схемой Горнера:

$$P_n(x) = (\dots((a_n x + a_{n-1}) x + a_{n-2}) x + \dots + a_1) x + a_0.$$

Например, для  $n=4$

$$P_4(x) = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0.$$

Словесно-формульная запись вычисления значения многочлена степени  $n$  по схеме Горнера может быть, например, следующая:

1.  $i := n$ .
2.  $S := a_i$ .
3.  $i := i - 1$ .
4.  $S := S \times x + a_i$ .
5. Если  $i \geq 1$ , то перейди в п. 3.
6. Стоп.

Здесь пп. 1, 2 и 6 выполняются по одному разу, а пп. 3—5 выполняются  $n$  раз. Этот алгоритм позволяет вычислять значения произвольного многочлена степени  $n$  ( $n \geq 1$ ) в любой точке  $x$ . При  $n=0$  он, к сожалению, неверен, так как получается  $S = a_0 x + a_{-1}$  вместо

$S = a_0$  ( $a_{-1}$  здесь вообще не определено). Чтобы исправить замеченный недостаток, запишем алгоритм несколько иначе.

1.  $i := n$ .
2.  $S := 0$ .
3.  $S := S \times x + a_i$ .
4.  $i := i - 1$ .
5. Если  $i \geq 0$ , то перейти к п. 3.
6. Стоп.

Пункты 1, 2 и 6 выполняются по одному разу, последовательность пп. 3—5 выполняется  $n+1$  раз. Этот алгоритм уже справедлив и для  $n=0$ , т. е. он более массовый, чем предыдущий. Алгоритм получился достаточно простым, наглядным и, что очень существенно, фактически не зависящим от степени многочлена. В самом деле, степень многочлена влияет лишь на п. 1 алгоритма и, естественно, на количество повторений цикла, т. е. пп. 3—5; чем выше степень многочлена, тем большее число раз повторится выполнение пп. 3—5.

Таким образом, в рассмотренных выше примерах число повторений цикла определяется, например, при вычислении суммы элементов последовательности — числом слагаемых; при нахождении максимума из  $n$  чисел — числом сравниваемых величин; при нахождении значения полинома — его степенью. Причем число повторений цикла известно до начала выполнения алгоритма и не зависит от самих величин, над которыми выполняются операции.

Составим теперь алгоритм решения задачи нахождения остатка  $R$  от деления  $a$  на  $b$  — двух целых неотрицательных чисел ( $b \neq 0$ ). При этом деление будем рассматривать как многократное вычитание. Алгоритм решения задачи можно представить в следующем виде:

1. Если  $a < b$ , то перейти к п. 4.
2.  $a := a - b$ .
3. Перейти к п. 1.
4.  $R := a$ .
5. Стоп.

Легко видеть, что цикл здесь составляют пп. 1—3. Число повторений этих пунктов заранее неизвестно и зависит от исходных величин — чисел  $a$  и  $b$ .

Рассмотрим еще одну задачу: найти решение уравнения  $f(x) = 0$ , где  $f(x)$  — заданная функция. Найти точные значения корней уравнения можно, как известно, лишь в исключительных случаях, обычно когда есть какая-либо простая формула для корней, выражающая их через известные величины. Так, даже для алгебраических уравнений (т. е. уравнений, в которых  $f(x)$  является алгебраическим многочленом степени  $n$ ) корни могут быть найдены через коэффициенты многочлена с помощью формул только в случае, когда степень многочлена не больше четырех; т. е. нет общих формул, использующих арифметические операции и извлечение квадратного корня, для вычисления

корней произвольного многочлена степени большей, чем четыре. Поэтому большое значение имеют методы приближенного вычисления корней уравнения  $f(x) = 0$ . Один из них — метод деления отрезка пополам.

Итак, пусть  $f(x)$  — непрерывная функция на отрезке  $[a, b]$  и известно, что уравнение  $f(x) = 0$  имеет на этом отрезке единственный корень  $x_0$ . Наша задача состоит в достаточно точном нахождении значения  $x_0$ .

В гл. 2 рассматривались различные численные методы решения уравнения  $f(x) = 0$ . Мы здесь предложим запись алгоритма метода деления отрезка пополам. Пусть функция  $f(x)$  непрерывна, имеет единственный корень на отрезке  $[a, b]$ , и на концах отрезка имеет разные знаки. Можно предположить, что  $f(a) < 0$  и  $f(b) > 0$ . Если же в действительности наоборот  $f(a) > 0$  и  $f(b) < 0$ , то можно точки  $a$  и  $b$  поменять ролями. Для этого достаточно выполнить следующие действия:  $l := a$ ;  $a := b$ ;  $b := l$ . Здесь  $l$  понадобилось, чтобы запомнить значение  $a$ , прежде чем оно будет уничтожено действием  $a := b$ . (Может возникнуть вопрос: а нельзя ли просто написать  $a := b$ ;  $b := a$ ? Нельзя, потому что в этом случае в результате получится, что  $a$  и  $b$  будут равны  $b$ .) В этих предположениях мы и дадим запись алгоритма отыскания корня  $x_0$ .

1. Если  $f(a) < 0$ , то перейти к п. 3.
2.  $l := a$ ;  $a := b$ ;  $b := l$ .
3.  $i := 0$ .
4.  $i := i + 1$ .
5.  $c_i := (a + b)/2$ .
6. Если  $f(c_i) = 0$ , то перейти к п. 12.
7. Если  $f(c_i) > 0$ , то перейти к п. 10.
8.  $a := c_i$ .
9. Перейти к п. 11.
10.  $b := c_i$ .
11. Если  $|b - a| \geq \varepsilon$ , то перейти к п. 4.
12.  $x_0 := c_i$ .
13. Стоп.

В п. 2 выполняется сразу три операции присваивания. Разумеется, их можно было записать отдельными пунктами. И наоборот, можно было объединить в один пункт все, что записано в пп. 2, 3.

В этом примере число повторений выделенной группы пунктов алгоритма заранее неизвестно и определяется как значениями  $a$ ,  $b$  и функцией  $f(x)$ , так и требуемой точностью результата  $\varepsilon$ . Причем ясно, что чем меньше значение  $\varepsilon$ , тем, вообще говоря, длительнее процесс нахождения приближенного значения корня. Кстати, для того, чтобы узнать, сколько же раз пришлось выполнить цикл для достижения заданной точности, достаточно посмотреть конечное значение переменной  $i$ , которое, кроме того, что является индексом у пере-

менной  $c_i$ , в сущности, играет еще и роль счетчика количества повторений цикла.

Этот алгоритм можно несколько упростить, если обратить внимание на то, что для всех точек разбиения  $c_i$  можно отводить одну и ту же переменную  $c$ .

1. Если  $f(a) < 0$ , то перейти к п. 3.
2.  $l := a$ ;  $a := b$ ;  $b := l$ .
3.  $c := (b + a)/2$ .
4. Если  $f(c) = 0$ , то перейти к п. 10.
5. Если  $f(c) > 0$ , то перейти к п. 8.
6.  $a := c$ .
7. Перейти к п. 9.
8.  $b := c$ .
9. Если  $|b - a| \geq \varepsilon$ , то перейти к п. 3.
10.  $x_0 := c$ .
11. Стоп.

Можно было бы и в этом случае узнать количество повторений цикла, если сохранить пп. 3 и 4 из первого варианта записи алгоритма. В этом случае переменная  $i$  играла бы только роль счетчика количества повторений цикла. До сих пор все наши примеры носили вычислительный характер. Но, как мы знаем, можно строить алгоритмы по обработке информации самого произвольного вида.

Приведем примеры обработки текстовой информации. Возьмем какое-нибудь слово на русском языке и поставим задачу заменить все буквы «о» в этом слове на буквы «а», а заодно и посчитать  $k$  — количество таких замен. Если в заданном слове буква «о» не встретится, то оставим слово без изменения, а  $k$  будем считать равным нулю.

Исходное слово состоит из букв  $b_1 b_2 b_3 \dots b_n$ , где  $b_i$  — любая буква русского алфавита (они могут совпадать). Тогда алгоритм решения нашей задачи можно записать следующим образом:

1.  $i := 1$ ;  $k := 0$ .
2. Если  $b_i \neq \text{«о»}$ , то перейти к п. 4.
3.  $b_i := \text{«а»}$ ;  $k := k + 1$ .
4.  $i := i + 1$ .
5. Если  $i \leq n$ , то перейти к п. 2.
6. Стоп.

Здесь поочередно просматриваются все буквы слова. Если очередная буква не совпадает с «о», то происходит переход к проверке следующей буквы (из п. 2 сразу в п. 4). Если же очередная буква оказалась буквой «о», то после п. 2 выполнится п. 3, где эта буква заменится на «а», и в счетчик количества таких замен добавится единица ( $k := k + 1$ ). Лишь после этого произойдет переход к проверке следующей буквы. И так происходит до тех пор, пока не будут просмотрены все буквы слова. За этим «следит» п. 5, где счетчик букв

сравнивается с  $n$  — количеством букв в слове. Если применить этот алгоритм к слову «ворон», то оно заменится на слово «варан» и количество замен букв будет равно двум, т. е.  $k=2$ .

Приведенный алгоритм замены всех вхождений одной буквы на другую является простейшим примером задачи обработки текстовой информации. В настоящее время ЭВМ, в особенности персональные компьютеры, очень широко используются для редактирования текстов, оформления документов различного назначения. Такая обработка производится по весьма сложным и интересным алгоритмам, которые представляют собой комбинации из таких простейших действий, как замена одной буквы или слова на другую букву или слово, как подсчет букв и т. д.

Советский ученый А. А. Марков в свое время предложил алгоритмическую систему, основной операцией в которой является подстановка в слово вместо одной группы букв другой группы букв; и высказал предположение, что в рамках такой системы можно выразить любой сколь угодно сложный алгоритм обработки данных.

Итак, на нескольких примерах мы продемонстрировали способ записи алгоритма, который можно было бы условно назвать словесно-формульным.

Словесно-формульная запись отличается от строгих языков описания алгоритмов для ЭВМ тем, что здесь допустим произвол в обозначениях и используемых словах.

Другим инструментом, часто используемым для записи алгоритмов, являются *блок-схемы*. Они служат для графического изображения структуры алгоритмов. Любая блок-схема представляет собой набор некоторых геометрических фигур или блоков, как правило, прямоугольников, ромбов, овалов. В этих фигурах можно записать любой текст, любую формулу, и вообще любую информацию о том, что надлежит сделать на данном этапе выполнения алгоритма. Последовательность действий в соответствии с блок-схемой указывается с помощью стрелок, соединяющих отдельные блоки и показывающих, какой блок и вслед за каким должен выполняться.

Из рассмотренных ранее алгоритмов видно, что очень часто встречаются разветвляющиеся процессы. В блок-схемах разветвлению соответствует логический блок, изображаемый обычно в виде ромба, внутри которого записывается некоторое условие. От логического блока ведут две стрелки — одна с надписью «да», другая — «нет». Если условие оказывается выполненным, то следующим по порядку будет блок, на который указывает стрелка с надписью «да», в противном случае очередным по порядку будет блок, на который отсылает стрелка с надписью «нет».

В качестве примера на рис. 4.1 приведена блок-схема рассмотренного выше алгоритма нахождения остатка  $R$  от деления двух целых чисел  $a \geq 0$  и  $b > 0$ .

Для удобства пояснений мы перенумеровали блоки. Здесь в блоке 1 проверяется условие  $a < b$ . В зависимости от того, выполняется оно или нет, осуществляется переход на один из блоков—2 или 3. При любых  $a$  и  $b$  алгоритм дает правильный результат. Только в случае, когда  $a$  меньше чем  $b$ , мы сразу же получим результат, перейдя от блока 1 к блоку 2, если же  $a$  не меньше чем  $b$ , то вслед за блоком 1

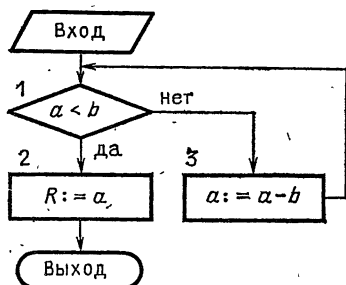


Рис. 4.1

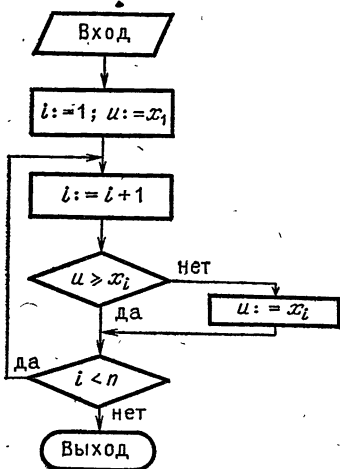


Рис. 4.2

выполняются действия, представленные блоком 3. Здесь произойдет присваивание  $a := a - b$  и затем будет снова проверяться условие  $a < b$  и т. д. Причем каждый раз из блока 3 в блок 1 мы будем приходиться с новым значением  $a$ , равным  $a - b$ . Так будет продолжаться до тех пор, пока, наконец, не выполнится условие  $a < b$ . Тогда от блока 1 попадем в блок 2 и выполнение алгоритма завершится.

Представим теперь в виде блок-схем три других рассмотренных ранее алгоритма: определение максимального из  $n$  заданных чисел (рис. 4.2), вычисления значения многочлена по схеме Горнера (рис. 4.3) и решение уравнения  $f(x) = 0$  (рис. 4.4).

Рассмотренные примеры позволяют сделать некоторые выводы о преимуществах и недостатках блок-схем. Основное преимущество — наглядность. Блок-схема позволяет отчетливо представить себе логические связи между частями алгоритма, проследить за последовательностью действий в алгоритме, проверить, все ли возможные варианты задачи нашли в нем отражение. Как правило, если алгоритм сложен, его пытаются представить в виде достаточно крупных блоков, чтобы выявить основные связи и сущность выполняемых действий. Затем уже крупные блоки можно разбить на более мелкие, т. е. составить для таких крупных блоков свои блок-схемы, проверить их логическую правильность, далее составить еще более мелкие блок-схемы и т. д. Так, в блок-схеме алгоритма решения уравнения  $f(x) = 0$  (рис. 4.4) в нескольких блоках происходит сравнение  $f(c)$  с нулем. Подразуме-

вается, что функция  $f(x)$  в точке  $c$  каким-то образом вычисляется. Если эта функция, например,  $P_n(x)$  — многочлен степени  $n$ , то для вычисления функции можно воспользоваться алгоритмом, предложенным на рис. 4.3, который стал бы частью алгоритма отыскания корня.

Может возникнуть вопрос, а нельзя ли сразу составить такую блок-схему сложного алгоритма, чтобы в ней отразились уже все детали. Конечно, можно. Но вряд ли целесообразно. В этом случае потеряется основное достоинство блок-схем — их наглядность. Трудно будет выявить неточности в алгоритме, т. е. исчезнут все те преимущества, ради которых и создавался язык блок-схем.

Мы здесь рассмотрели лишь самые примитивные способы записи алгоритмов. Они хороши на первом этапе составления программ, когда нужно представить алгоритм в целом, осмыслить его логические связи, посмотреть все возможные варианты.

Для того, чтобы поручить машине решение задачи по выбранному алгоритму, нужно записать его уже на каком-то другом языке, таком,

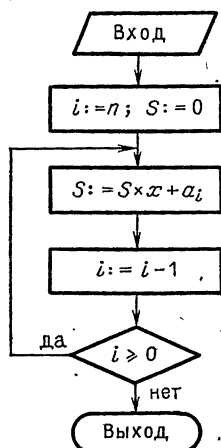


Рис. 4.3

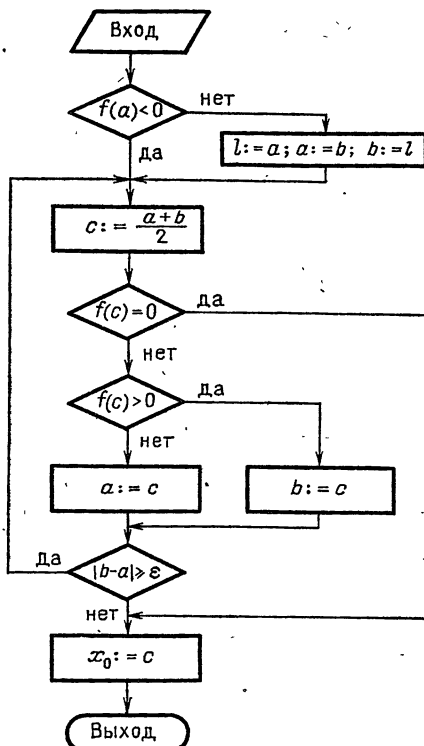


Рис. 4.4

чтобы его уже могла понять и выполнить машина. У каждой ЭВМ есть свой внутренний язык, и можно, конечно, написать программу на этом языке. Для машины такая программа понятна, она ее может сразу же выполнять. Так и писали программы в те времена, когда появились первые ЭВМ. Однако, как мы в дальнейшем убедимся, для

программиста такой способ весьма неудобен. Составление такой программы — процесс трудоемкий, программа выглядит громоздкой, разобраться в ней сложно не только постороннему человеку, но зачастую даже самому автору; очень трудно отыскать в ней ошибки. Поэтому стремятся создать такие способы записи алгоритмов, чтобы программа была написана на языке, максимально приближенном к обычному человеческому, с использованием принятой математической символики.

В то же время программа должна пониматься четко и однозначно, чтобы любой исполнитель мог ее воспринимать одинаково и именно так, как мыслил автор программы. Тогда работу по выполнению алгоритма можно поручить и электронной вычислительной машине. Разумеется, ЭВМ должна для этого сначала перевести программу на свой внутренний язык. Это, конечно, потребует дополнительного машинного времени. Но эти затраты окупаются за счет значительного облегчения труда программиста при составлении программ.

## § 4.2. Понятие о символьном кодировании

В предыдущих главах мы познакомились с некоторыми способами записи алгоритмов, в частности с представлением их в виде последовательности команд ЭВМ. Отмечалось, что такое представление алгоритма, т. е. запись его на языке машины, является для ЭВМ последовательностью указаний, каждое из которых реализуется соответствующей командой из фиксированного набора команд ЭВМ.

Мы убедились в том, что каждый тип вычислительных машин имеет свой вид команд. Системы команд различных ЭВМ могут отличаться форматом команд, т. е. количеством адресов в команде, количеством разрядов, отводимых под каждый адрес, назначением информации, задаваемой в адресах, набором операций, которые может выполнять машина, и обозначением этих операций. Другими словами, каждая ЭВМ имеет свой машинный язык. Поэтому человек, составляющий программу, в которой необходимо использовать все возможности какой-либо конкретной вычислительной машины, должен хорошо знать язык этой ЭВМ и его особенности.

Очень редко возникает необходимость писать программы в кодах конкретной машины. Это трудоемкий, утомительный процесс. Программа получается, как правило, громоздкой, труднообозримой, содержит немало ошибок, выявить и устранить которые весьма непростое. Обычно это требует значительно больше времени и усилий, чем написание самой программы. Тем более, что внося исправления в программу, программист часто невольно допускает новые ошибки.

Немало трудностей приходится преодолевать и при внесении изменений в программу, если по какой-либо причине понадобится модифицировать сам алгоритм решения задачи. Если же у кого-нибудь появится необходимость разобраться в чужой машинной программе,



то это оказывается уже практически безнадежным делом — легче заново программу составить самому. Даже автор машинной программы по прошествии достаточно большого промежутка времени уже не всегда может толком понять ее.

Как мы уже отмечали, на первом этапе развития вычислительной техники программы писались только на машинных языках. Многообразие типов вычислительных машин, а следовательно, и различных машинных языков делало практически невозможным перенос программ с одних машин на другие. Приходилось для каждого типа ЭВМ создавать свой набор программ для решения стандартных задач. При замене одних машин на другие все эти программы становились абсолютно ненужными и надо было писать новые программы.

Таким образом, назрела необходимость в разработке новых способов и средств записи алгоритмов и программ. Поначалу пытались упростить процесс составления программ на машинном языке. Составлялись специальные программы, которые были призваны облегчить труд программиста. Так, например, при решении большой задачи можно разбить алгоритм ее решения на отдельные части, и программы для каждой части писать независимо. Это, конечно, гораздо проще, чем писать программу для всего сложного алгоритма. А затем специальная программа объединит получившиеся фрагменты программы в единое целое.

Создавались и так называемые отладочные программы, которые помогали находить и устранять ошибки в отлаживаемых программах. Тем не менее, основные трудности программирования, связанные с машинным языком, оставались. Для существенного облегчения труда программиста необходимы были новые средства для написания программ.

В математике давно существует способ записи алгоритмов в виде расчетных формул. По существу каждую формулу можно рассматривать как запись алгоритма вычисления значений искомой величины. Мы часто говорим о языке математических формул, понимая под этим правила записи различного рода выражений, которые диктуют нам последовательность и правила выполнения расчетов. Любой грамотный человек понимает этот язык формул и может выполнить указанные в формулах расчеты. Обратим внимание на следующее важное обстоятельство. Формулы состоят из букв и знаков операций. Чтобы выполнить вычисления, заданные формулой, надо вместо букв подставить в формулу конкретные числовые значения и получить результат. Буквы тем самым можно назвать символами, заменяющими какие-то конкретные понятия, в данном случае — числа. Изобретение алгебры является величайшим достижением математики, которое позволило не только представлять правила вычисления в компактном, легко обозримом виде, но и проводить различного рода преобразования формул, устанавливая законы и следствия, оперируя только с символически-

ми записями. В этой связи можно сказать, что языки программирования являются дальнейшим развитием идеи представления алгоритмов в виде символической, формульной записи.

Однако, если в элементарной математике символами обозначаются, как правило, числа, в языках программирования таким образом представляются и многие другие объекты и понятия. Запись алгоритма на языке программирования должна быть обзримой и легкой для понимания за счет ее близости к записи в виде формул, близости к естественному языку. В то же время необходимы достаточно строгие правила, определяющие язык, для того, чтобы любая запись на таком языке понималась точно и однозначно не только ее автором, но и любым другим человеком, а также ЭВМ.

Мы уже знаем, что любое десятичное число может быть закодировано двоичными символами. Аналогично можно и любую запись, состоящую из букв, знаков и слов алгоритмического языка закодировать двоичными цифрами. Если затем ввести закодированную информацию в память ЭВМ, то, поскольку запись эта понимается однозначно, можно поручить самой ЭВМ расшифровать ее. Для решения этой задачи машина должна иметь специальную программу, называемую транслятором, которая осуществит перевод записи алгоритма на язык данной машины. Процесс этот очень не прост. Транслятор должен уметь анализировать каждое предложение на таком языке, ставить ему в соответствие некоторый набор команд машины. Наряду с этим, транслятор должен в процессе перевода найти имеющиеся ошибки в записи алгоритма, дать их точную и по возможности детальную диагностику, которая помогла бы программисту быстро устранить эти ошибки.

При создании языков программирования идут, как правило, по двум направлениям. Первое направление — разработка символических языков, которые были бы максимально приближены к машинным командам конкретных ЭВМ, а второе — создание языков ни в коей мере не ориентированных на конкретные ЭВМ, или как говорят, машинно-независимых языков.

В данной главе мы остановимся на первом из этих двух направлений и поведем разговор о машинно-ориентированных языках. Программа на таком языке по своему виду напоминает программу на языке машины. Однако в ней при записи используются символические имена ячеек, где хранятся значения переменных, вместо цифровых кодов операций, пишутся легко запоминающиеся условные обозначения. Такой машинно-ориентированный язык называют автокодом или ассемблером. Запись программы на автокоде представляет собой последовательность предложений или строк автокода, которые пишутся с соблюдением определенных правил.

Программа, составленная на автокоде, вводится в память ЭВМ. Прежде чем выполнить программу, записанную на автокоде, трансля-

тор переводит ее на язык машины. Естественно, что программы, организующие процесс трансляции, также должны находиться в памяти ЭВМ. Введенный в память закодированный текст программы является для программы трансляции исходными данными, которые преобразуются в программу на машинном языке. При этом в процессе трансляции могут обнаружиться ошибки в записи программы, и транслятор выдает о них сообщение. Ошибки необходимо устранить, после чего снова поручить транслятору перевод программы. Если ошибок больше нет, то можно приступить к выполнению программы.

Обычно каждой машинной команде конкретной ЭВМ соответствует одна строка, или предложение автокодной программы. В этом случае говорят, что автокод имеет тип 1 : 1 (один к одному). Однако существуют и такие автокоды, где одному автокодному предложению может соответствовать и последовательность из нескольких машинных команд.

При трансляции автокодного предложения транслятор заменяет символический (мнемонический) код операции соответствующим цифровым кодом, а вместо обозначения переменной вставляет отведенный тем же транслятором для нее машинный адрес. Тем самым автокодное предложение становится обычной машинной командой.

При написании программы на автокоде программист может наиболее полно учитывать и использовать все возможности и особенности данной конкретной ЭВМ. Поэтому программа после трансляции с автокода получается достаточно быстройдействующей и не слишком громоздкой, разумеется, насколько это возможно для решаемой задачи и данной конкретной ЭВМ.

Программа на автокоде значительно нагляднее, чем программа в машинных кодах, состоящая из последовательностей цифр. Обозначения адресов здесь могут совпадать с названиями переменных программируемой задачи, коды операций мнемоничны и достаточно легко запоминаются. Например, если нужно проделать вычисления по известной из курса физики формуле  $V = S/t$ , где  $V$  — скорость движения,  $S$  — пройденный путь, а  $t$  — затраченное время, то реализующее это вычисление автокодное предложение для трехадресной машины выглядит так:

Д S T V

Здесь Д — условное обозначение операции деления.  $S$ ,  $T$  и  $V$  — имена ячеек, в которых содержатся переменные, участвующие в вычислениях.

Очевидно, что всем кодам операций можно дать легко запоминающиеся условные обозначения, например, С — сложение, В — вычитание, У — умножение, ВА — вычитание абсолютных величин и т. д.

В автокодной программе значительно легче, чем в программе, написанной в машинных кодах, отыскиваются и устраняются ошибки.

При необходимости вставить или удалить команду в машинной программе приходится менять номера ячеек многих команд, а в связи с этим и содержимое адресных частей некоторых команд, что часто приводит к путанице: что-то изменилось, а что-то нет. В автокоде об этом можно не заботиться. Если мы вставили или удалили какое-либо предложение, то при последующей трансляции команды будут расставлены в нужном порядке и ошибки не будет.

Мы здесь не ставим себе цель подробно и точно описать какой-либо конкретный автокод—это было бы слишком сложно сделать в рамках данной книги, да и вряд ли целесообразно. Мы ограничимся лишь рассмотрением самых основных черт машинно-ориентированных языков, оставив в стороне не только тонкости, но даже зачастую и довольно важные его аспекты. В сущности, то, о чем у нас пойдет речь, нельзя назвать автокодом—это некоторое представление о языке символического кодирования.

Программист, который собирается программировать на машинно-ориентированном языке, должен иметь достаточно полное и точное представление об ЭВМ, для которой составляется программа. Он должен знать систему команд, знать какие действия выполняет ЭВМ, исполняя каждую команду программы. Он должен знать, как в машинной памяти располагаются данные и команды, как они адресуются. Он должен учитывать детали, связанные с представлением чисел, положением знака числа, способом округления результатов. В некоторых случаях он должен знать времена исполнения команд, так как общее время выполнения всей программы, так же как и точность полученного результата может зависеть от выбранного порядка вычислений. Например, безразлично с этой точки зрения, что запрограммировать:  $a : \pi$  или  $a \times 1/\pi$ . Если позаботиться о том, чтобы заготовить заранее константу, равную  $1/\pi$ , то последнее выражение будет вычислено заметно быстрее.

В данном параграфе мы постараемся дать представление о системе команд и способах представления данных в некой трехадресной ЭВМ, напоминающей машину М-20.

Мы выбрали именно эту ЭВМ по следующим соображениям. Машина М-20 в свое время сыграла большую роль в становлении советской вычислительной техники. Она, являясь оригинальной разработкой, превосходила по многим параметрам зарубежные типы ЭВМ аналогичного назначения.

При разработке системы команд этой ЭВМ учитывалось удобство «ручного» неавтоматизированного программирования на уровне машинных команд. В те времена это было оправдано, так как автоматизация программирования только еще зарождалась. М-20 имела наиболее естественную трехадресную систему команд. В команде указывался код операции, адреса двух операндов и адрес результата. Кроме того, в ее структуре были предусмотрены средства, позволявшие организо-

вывать циклы, решать задачи, связанные с циклической обработкой массивов данных. Особенно просто и естественно программировались вычисления по формулам.

### § 4.3. Описание системы команд машины

Оперативная память нашей несколько упрощенной по сравнению с реальной машины состоит из ячеек одинаковой длины, т. е. содержащих одинаковое число двоичных разрядов. Мы пока не оговариваем здесь общее число ячеек в оперативной памяти машины. Скажем лишь, что оно ограничено.

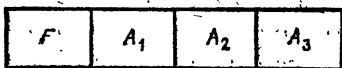


Рис. 4.5

В каждой ячейке оперативной памяти нашей машины может располагаться либо одно число, либо одна команда. В том случае, когда в ячейке расположена команда, она рассматривается устройством управления как состоящая из отдельных частей (так называемых полей). Каждое поле имеет свой смысл — в нем указывается в двоичном закодированном виде либо информация о выполняемой операции — код операции, либо информация об адресах операндов — коды адресов.

Схема распределения полей в ячейке, содержащей команду, представлена на рис. 4.5. Здесь  $F$  — поле кода операции;  $A_1$  — поле адреса первого операнда;  $A_2$  — поле адреса второго операнда;  $A_3$  — поле третьего операнда (результата).

Поле  $F$  обычно называют функциональным полем. В нем содержится правило выполнения команды, в частности, указывается, какую операцию следует выполнить.

Если в поле  $F$  указан код ( $\theta$ ) одной из арифметических операций (сложение, вычитание, умножение, деление), то, как мы уже знаем, машиной эта команда выполняется по следующему правилу:

- первый операнд выбирается из памяти по адресу, номер которого записан в поле  $A_1$ ;

- второй операнд выбирается из памяти по адресу, определяемому полем  $A_2$ ;

- с выбранными операндами производится действие, указанное в поле  $F$ ;

- результат отправляется в ячейку памяти по адресу, указанному в поле  $A_3$ .

Пусть, например, нам нужно выполнить операцию сложения:

$$z := x + y$$

Чтобы записать команду в машинных кодах, надо указать цифровой код операции и адреса операндов, т. е. номера ячеек, в которых хранятся значения  $x$  и  $y$ , а также номер ячейки, в которую следует поместить результат, т. е. полученное значение  $z$ .

Пусть для определенности  $x$  хранится в ячейке с номером 1200,  $y$  — в ячейке 3450, а результат  $z$  нужно поместить в ячейку 2000. Если код операции сложения 01, то команда на языке машины, реализующая данную операцию, выглядит так:

01 1200 3450 2000

$F \quad A_1 \quad A_2 \quad A_3$

В действительности, как мы уже знаем, любое машинное слово записывается только с помощью нулей и единиц. Однако, как известно, любую тройку двоичных цифр можно однозначно представить одной восьмеричной цифрой. Это мы сейчас и сделали, выписав код операции и адреса ячеек в восьмеричном коде. Запись команды получилась втрое компактнее по сравнению с двоичной записью.

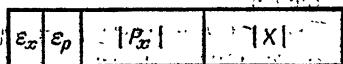
Под поле каждого адреса в машине отводится 12 двоичных разрядов. Тем самым в каждом адресе можно записать 12-разрядный двоичный или, что то же самое, 4-разрядный восьмеричный номер ячейки. Поэтому мы можем в каждом адресе указать номер от (0000)<sub>8</sub> до (7777)<sub>8</sub>.

Числа в машине представляются в двоичной системе в форме с плавающей запятой, т. е.

$$x = X \cdot 2^{P_x},$$

где  $X$  — мантисса числа,  $P_x$  — порядок.

В машине разряды ячейки при записи в нее числа распределяются следующим образом:



Здесь  $\varepsilon_x$  — знак числа,  $\varepsilon_p$  — знак порядка («+» изображается цифрой 0, «-» — цифрой 1), ( $P_x$  — порядок числа) ( $0 \leq |P_x| \leq (77)_8 = = (63)_{10}$ ),  $X$  — мантисса.

Обычно машина производит операции над нормализованными числами (т. е.  $1/2 \leq |X| < 1$ ) и результат тоже выдает в нормализованном виде. Диапазон представления нормализованных чисел для нашей ЭВМ определяется соотношением

$$2^{-64} \leq |x| < 2^{63}, \quad \sim 10^{-19} < |x| < \sim 10^{19}.$$

Для более полного описания ЭВМ нужно, конечно, сказать о том, как распределены поля ячейки не только в командах арифметики, но и в других, так называемых, управляющих командах. Следует так же сказать и о командах ввода-вывода, командах обращения к внешним запоминающим устройствам. Все такие машинные команды находят свое отражение в языке символьного кодирования. О машинном представлении этих команд мы будем рассказывать далее по мере

надобности для дальнейшего изложения. Пока что нам будет достаточно приведенной выше информации.

Мы рассмотрели машинную кодировку команды, вычислявшей по формуле

$$z := x + y.$$

Эту же команду запишем на языке символьного кодирования. Для этого номера ячеек, в которых хранятся значения  $x$  и  $y$ , обозначим соответственно символами  $X$  и  $Y$ , номер ячейки, в которую мы хотим поместить результат, обозначим символом  $Z$ . Наконец, код операции сложения обозначим символом  $S$ . Тогда команда запишется так:

$S \ X \ Y \ Z \mid z := x + y$

(справа от команды мы поместили пояснения к ней).

Совершенно очевидно, что такое представление команды нагляднее и проще, чем запись в цифровом машинном коде. В этом убеждает хотя бы то, что здесь не нужно запоминать цифровые коды операций, знать в каких ячейках размещаются различные данные. Из написания команды сразу ясно, какая это операция и над какими объектами она производится. Не удивительно поэтому, что при изложении приемов программирования на трехадресной ЭВМ мы в дальнейшем будем пользоваться только символьными обозначениями.

Заметим, что те программы, которые мы будем составлять, будут всегда фрагментами некоторых более общих программ, хотя бы потому, что мы не будем касаться вопросов ввода и вывода информации.

#### § 4.4. Программирование вычислений по формулам

ЭВМ первых поколений использовались в основном для решения задач вычислительного характера. В такого рода задачах часто приходится программировать вычисления по формулам. Для этого машина должна уметь выполнять ряд арифметических операций. Эти операции она осуществляет над числами, представленными в памяти ЭВМ в нормализованном виде. Результат операции также получается в нормализованном виде.

В описаниях ЭВМ перечень команд часто задают таблицей (табл. 4.1).

При описании команд для реальных ЭВМ точно указываются все ситуации, которые могут возникнуть при их выполнении. Например, если результат выполнения операции таков, что для размещения его мантиссы недостаточно количества отведенного под нее разрядов в ячейке, то автоматически происходит округление результата, а затем его нормализация. Возможно, что результат операции окажется больше по абсолютной величине, чем самое большое число, представимое в ЭВМ. В этом случае происходит аварийный останов по пере-

## Арифметические операции

Название операции	$\theta$ -символический код	Содержание команды
Сложение	С	$z := x + y$
Вычитание	В	$z := x - y$
Вычитание абсолютных величин	ВА	$z :=  x  -  y $
Умножение	У	$z := x \times y$
Деление	Д	$z := x / y$

полнению. Так будет, например, если в результате арифметической операции получится число, превышающее  $2^{63}$ .

При программировании на языке символьного кодирования все подобные ситуации должны быть учтены. Программист должен их хорошо знать. В противном случае ему будет очень сложно отлаживать программу. Он не сможет понять почему, например, ЭВМ останавливается на участке программы, в котором на первый взгляд нет ничего сложного. Причем может получиться, что при одних исходных данных все идет нормально, а при других получаются останова. Отыскать этому причину без знания соответствующих «машинных» тонкостей практически невозможно.

Пользуясь перечисленными выше в таблице операциями, мы уже можем составлять программы для вычислений по простым формулам. Напишем, например, программу вычисления:

$$z = (x - y + 1) / (x^2 + y^2).$$

Последовательность действий, содержащих по одной арифметической операции, для решения задачи на языке символьно-формульной записи может быть такой:

$$r_1 := x - y$$

$$r_2 = r_1 + 1$$

$$r_3 = x \times x$$

$$r_4 := y \times y$$

$$r_5 := r_3 + r_4$$

$$z := r_2 / r_5$$

Здесь  $r_1, r_2, r_3, r_4, r_5$  обозначают промежуточные результаты.

Будем считать, что величины  $x, y$  и  $1$  находятся в некоторых ячейках памяти ЭВМ, и, как принято в символьном кодировании, вместо номеров этих ячеек мы укажем в командах их символические обозначения, соответственно —  $X, Y$  и  $ED$ . Промежуточные результаты поместим в ячейки, обозначенные символами  $R1, R2, R3, R4$  и  $R5$



(обычно такие ячейки называют рабочими ячейками). Результат  $z$  поместим в ячейку  $Z$ .

В этом случае программа в символических обозначениях, реализующая наши вычисления, принимает вид

В	X	Y	R1	$r_1 := x - y$
С	R1	ЕД	R2	$r_2 := r_1 + 1$
У	X	X	R3	$r_3 := x \times x$
У	Y	Y	R4	$r_4 := y \times y$
С	R3	R4	R5	$r_5 := r_3 + r_4$
Д	R2	R5	Z	$z := r_2 / r_5$

Все команды находятся в памяти машины в последовательно расположенных ячейках и выполняются одна за другой в том порядке, в каком они написаны. Справа от вертикальной черты мы поместили пояснения к программе, которые, впрочем, мало отличаются по наглядности от вида самих команд языка автокода.

В нашей программе можно уменьшить количество используемых рабочих ячеек, учитывая тот факт, что значение содержимого некоторых из них после выполнения соответствующих команд в дальнейшем не используется. Так, вычислив  $r_1 := x - y$ , мы затем прибавляем к найденному значению единицу, и само значение  $(x - y)$  нам уже больше не понадобится. Поэтому вместо новой рабочей ячейки  $R2$  можно использовать ячейку  $R1$ . Аналогично вместо ячеек  $R4$  и  $R5$  можно использовать ячейку  $Z$ , предназначенную для результата.

В этом случае программа будет выглядеть следующим образом:

В	X	Y	R1	$r_1 := x - y$
С	R1	ЕД	R1	$r_1 := r_1 + 1$
У	X	X	R2	$r_2 := x \times x$
У	Y	Y	Z	$z := y \times y$
С	R2	Z	Z	$z := r_2 + z$
Д	R1	Z	Z	$z := r_1 / z$

Еще раз подчеркнем, что все наименования операндов в программе есть символические обозначения номеров ячеек, в которых размещаются величины, участвующие в операциях. В процессе трансляции эти условные обозначения будут заменены на истинные номера ячеек, а сама программа будет размещена в подряд стоящих ячейках памяти машины, начиная с некоторой ячейки, которую по своему усмотрению задает программист с помощью специального указания транслятору.

#### § 4.5. Программирование разветвляющихся процессов

В предыдущих главах мы имели возможность убедиться, что далеко не все алгоритмы, даже вычислительного типа, имеют линейный, строго последовательный характер. Очень часто приходится выполнять

ту или иную последовательность действий в зависимости от промежуточных результатов.

Пусть, например, надо найти величину  $y = \max\{x_1, x_2\}$ . Блок-схема алгоритма решения этой задачи представлена на рис. 4.6.

Мы видим, что в зависимости от того, какое из двух чисел больше —  $x_1$  или  $x_2$ , осуществляется переход на различные блоки алгоритма, т. е. происходит разветвление процесса вычислений. Подобных примеров может быть сколь угодно много. Как же такие алгоритмы реализовать в виде программ для ЭВМ?

Вспомним, что порядком выполнения команд на ЭВМ управляет специальный регистр «счетчик команд». Содержимое этого регистра задает номер ячейки, слово из которой рассматривается устройством управления ЭВМ как команда, подлежащая выполнению. Для того, чтобы очередная команда программы сработала в нужный момент, необходимо, чтобы к этому моменту в счетчике команд оказался номер именно той ячейки, в которой эта команда содержится.

Как уже рассказывалось, выполнение каждой команды, если это не специальная команда изменения содержимого счетчика команд, завершается тем, что счетчик команд автоматически увеличивается на единицу. Это означает, что после выполнения данной команды машина перейдет к исполнению команды, стоящей в следующей по номеру ячейке. Для программирования разветвляющихся процессов такой естественный порядок приходится нарушать.

В самом деле, память ЭВМ линейна, т. е. все ячейки нумеруются

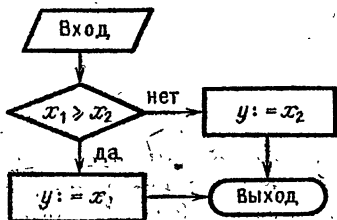


Рис. 4.6

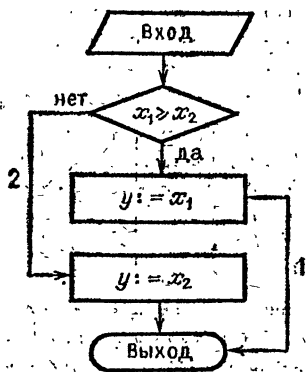


Рис. 4.7

последовательно одна за другой. И, если мы размещаем подряд в памяти машины части программы, соответствующие различным ветвям алгоритма, то при выполнении какого-либо определенного алгоритмом условия должна выполняться одна часть программы, и не должна выполняться другая, т. е. эту другую часть надо обойти, и, следовательно, нарушить естественный порядок выполнения команд.

Например, блоки программы, реализующей алгоритм вычисления значения  $y = \max\{x_1, x_2\}$ , могут располагаться в памяти ЭВМ последовательно (рис. 4.7).

Мы видим, что при выполнении условия  $x_1 \geq x_2$  сразу же идет присваивание результата переменной  $y$  (т. е.  $y := x_1$ ). Вслед за этой частью программы в памяти ЭВМ следуют ячейки, в которых (если это одна ячейка, то — в которой) располагается команда засылки в качестве результата значения  $x_2$  (т. е.  $y := x_2$ ), а этого при  $x_1 \geq x_2$  делать как раз и не нужно. Значит блок, а следовательно и соответствующую часть программы, где происходит данное присваивание, необходимо обойти, что на нашем рисунке показано стрелкой 1. Если же условие  $x_1 \geq x_2$  не выполнено, то обойти нужно ту часть программы, которая переменной  $y$  присваивает значение  $x_1$ , и вслед за этим идти на выполнение присваивания  $y := x_2$ . Этот обход показан на рисунке стрелкой 2.

Из сказанного следует, что необходимы операции, которые бы могли по желанию программиста произвольным образом менять содержимое счетчика команд. Такие операции предусмотрены в каждой ЭВМ и называются командами перехода.

Простейший вид команды перехода — команда безусловного перехода, символическое обозначение которой ПБ (переход безусловный). А сама команда в автокоде имеет вид

ПБ — A2 —

Прочерками мы здесь и далее будем обозначать поля команды, содержащие все нули или поля, содержимое которых не влияет на выполнение операции.

В результате выполнения этой команды в счетчике команд окажется номер ячейки A2, независимо от того, что в счетчике находилось раньше. Поэтому следующая команда для исполнения будет взята из ячейки с номером A2.

Наличие такой команды в системе команд ЭВМ позволяет располагать в памяти ЭВМ программы, соответствующие различным блокам алгоритма в произвольном порядке, и, как мы увидим, это очень важно со многих точек зрения.

Команда, на которую мы хотим передать управление, должна быть, как говорят, помечена, т. е. ей нужно дать какое-то имя, или метку. Это имя в автокод-программе (так мы для краткости будем называть программу на автокоде) должно быть написано перед данной командой. А между именем и командой необходимо поставить знак двоеточие. Например,

L1 : B X Y Z

Здесь L1 служит именем данной команды. В машинном коде этому имени соответствует номер ячейки памяти, в которой реально располагается эта команда.

Для перехода к команде с именем L1 достаточно выполнить команду

ПБ — L1 —

И независимо от места расположения в памяти команды перехода следующей за ней будет выполнена команда, помеченная меткой L1.

Прежде чем перейти к рассказу о других командах перехода, мы вернемся к рассмотренным ранее операциям арифметического типа. Некоторые из этих операций кроме своего основного результата вырабатывают еще специальный признак, который мы обозначим символом  $\omega$ . Он может принимать лишь два значения 0 или 1.

В табл. 4.2 указано, при каком условии соответствующая операция вырабатывает признак  $\omega$ , равный 1 (в противном случае вырабатывается признак  $\omega=0$ ).

Т а б л и ц а 4.2

Название операции	Символический код	Результат	$\omega=1$
Сложение	С	$z := x + y$	$z < 0$
Вычитание	В	$z := x - y$	$z < 0$
Вычитание абсолютных величин	ВА	$z :=  x  -  y $	$z < 0$

Таким образом, если результат указанных операций окажется отрицательным, то вырабатывается признак  $\omega=1$ , в противном случае будем иметь  $\omega=0$ .

Для разветвления вычислительного процесса в наборе машинных операций есть специальная операция условного перехода. С ее помощью осуществляют выбор дальнейшего пути вычислений в зависимости от текущего значения признака  $\omega$ , т. е. от характеристики результата предыдущей операции.

После выполнения такой команды содержимое счетчика команд либо, как обычно, увеличивается на единицу, и тогда выполняется команда из следующей по порядку ячейки, либо в счетчик команд занесется номер ячейки, указанный во втором адресе команды перехода, а это может быть любая ячейка памяти машины, содержимое которой будет воспринято устройством управления как команда, подлежащая исполнению. Заметим, что сама команда условного перехода признак  $\omega$  оставляет без изменения. Одной из таких команд является команда П1 (переход по единице):

П1 — А2 —

Здесь П1 — обозначение операции условного перехода. Работает команда следующим образом: при  $\omega=0$  сохраняется естественный порядок выполнения команд, а при  $\omega=1$  происходит переход к выполнению команды из ячейки А2.

Аналогично можно ввести команду с кодом операции П0 (переход по нулю).

Записывается эта команда следующим образом:

П0 — A2 —

Иметь ту и другую команду перехода в списке команд машины удобно и во многих реальных ЭВМ они действительно существуют. Теперь для того, чтобы написать программу нахождения  $y = \max \{x_1, x_2\}$  нам понадобится еще операция пересылки содержимого одной ячейки в другую:

П A1 — A3

Здесь П — обозначение операции пересылки. Если  $a_1$  — содержимое ячейки A1, а  $a_3$  — содержимое ячейки A3, то по данной команде произойдет присваивание  $a_3 := a_1$ . При этом содержимое ячейки A1 не меняется.

Введем еще команду, результатом работы которой является прекращение выполнения программы

СТОП — — —

Предложим вариант программы, соответствующей блок-схеме алгоритма, изображенной на рис. 4.7:

В	X1	X2	R
П1	—	L1	—
П	X1	—	Y
ПБ	—	L2	—
L1: П	X2	—	Y
L2: СТОП	—	—	—

Рассмотрим подробно, как выполняется наша программа. Первая команда служит для выяснения вопроса, что больше —  $x_1$  или  $x_2$ . Заметим, что само по себе значение разности  $x_1 - x_2$  нас совершенно не интересует, нам важно лишь знать, какое из двух чисел больше. Поэтому первую команду программы можно было бы записать и так:

В X1 X2 —

Заметим, кстати, что здесь можно было бы подумать, что результат операции окажется в ячейке с нулевым номером. В действительности конструкция многих машин предусматривает, что в нулевой ячейке всегда содержится число нуль, и записать в эту ячейку ничего нельзя. Поэтому и результат нашей операции вычитания тоже не будет записан в нулевую ячейку. Однако, признак  $\omega$  все равно вырабатывается.

Если окажется, что  $x_1$  меньше, чем  $x_2$ , то разность между ними отрицательна и вырабатывается признак  $\omega = 1$ . Тогда следующая команда:

П1 — L1 —

осуществит переход на команду с меткой  $L1$ . По этой команде в качестве результата в ячейку  $Y$  перешлется максимальное значение. В данном случае  $x_2$  (т. е.  $y := x_2$ ).

Далее естественным порядком выполняется следующая команда — СТОП. Имеющиеся в программе команды:

П  $X1 - Y$

ПБ —  $L2$  —

в данном случае будут обойдены и не выполнятся.

Рассмотрим теперь второй случай. После выполнения команды вычитания выяснилось, что результат неотрицательный ( $x_1 \geq x_2$ ), и, следовательно, выработается признак  $\omega = 0$ . Тогда команда

ПБ —  $L1$  —

не нарушит естественный порядок выполнения команд и выполнится следующая по порядку команда:

П  $X1 - Y$

которая зашлет в ячейку  $Y$  максимальное из двух сравниваемых чисел, в данном случае  $x_1$ .

По окончании работы этой команды счетчик команд увеличится на единицу и выполнится следующая команда:

ПБ —  $L2$  —

которая заставит машину обойти команду, помеченную меткой  $L1$  и передаст управление команде с меткой  $L2$ , т. е. команде останова, по которой завершится выполнение программы.

Что произошло бы, если бы мы не поставили команду:

ПБ —  $L2$  —

Тогда в случае, когда выполнено условие  $x_1 \geq x_2$ , машина сначала бы выполнила команду

П  $X1 - Y$

в результате которой произошло бы присваивание  $y := x_1$ , а затем команду

П  $X2 - Y$

В ячейку  $Y$  в качестве результата было бы занесено значение  $x_2$ . В итоге оказалось бы, что  $y = x_2$ , хотя максимальным является значение  $x_1$ . Следовательно, мы бы получили неверный результат.

Заметим, что в символьном кодировании есть возможность ссылаться на какую-либо ячейку, не давая ей имя или метку, а указывать на нее, отсчитывая ее адрес относительно какой-то уже введенной в программе метки.

Например, если есть машинное слово с меткой  $M$ , то можно сослаться на следующее слово, указав адрес  $M+1$ , можно сослаться на предыдущее слово, указав адрес  $M-1$ , можно сослаться и на да-

леко отстоящее слово, скажем,  $M+100$ . Но в последнем случае есть опасность, что нам понадобится изменить программу на участке между командами, хранящимися в ячейках  $M$  и  $M+100$ . Например, вставить новые команды или выбросить какие-либо старые, в результате чего количество команд изменится. Тогда ячейка, которая ранее находилась на расстоянии в 100 ячеек от ячейки  $M$ , будут уже совсем на другом месте, а на этом месте окажется соответственно другое слово, и ссылка на него будет уже ошибочной. А значит понадобится ссылку на ячейку  $M+100$  изменить на новую. Поэтому подходить к использованию этой возможности следует осторожно. Гораздо проще в таком случае поставить перед соответствующим словом новую метку. Однако, когда опасность ошибки не грозит, рассмотренный выше прием вполне приемлем. Так нашу программу по отысканию максимума двух чисел можно записать следующим образом:

В	X1	X2	—	$x_1 < x_2?$
П1	—	L	—	
П	X1	—	Y	$y := x_1$
ПБ	—	L+1	—	
L: П	X2	—	Y	$y := x_2$
СТОП	—	—	—	

Здесь в команде безусловного перехода мы по второму адресу написали  $L+1$ , и произойдет переход на команду, следующую за командой с меткой  $L$ .

Команды с кодами П0 и П1 разветвляют ход вычислений по двум направлениям. А если ветвей в алгоритме больше? Покажем на примере, что и в этом случае никаких принципиальных трудностей не возникнет.

Рассмотрим алгоритм вычисления значения функции

$$y = \text{sign } x = \begin{cases} 1, & \text{если } x > 0, \\ 0, & \text{если } x = 0, \\ -1, & \text{если } x < 0. \end{cases}$$

Блок-схема алгоритма представлена на рис. 4.8.

Составим программу на автокоде, реализующую предложенный алгоритм:

В	X	—	—	$x \geq 0 ?$
П0	—	L	—	
В	—	ЕД	Y	$y := -1$ (так как $x < 0$ )
ПБ	—	M	—	
L: В	—	X	—	$-x \geq 0$ ( $x=0$ или $x < 0$ )
П1	—	M-1	—	
П	—	—	Y	$y := 0$ (так как $x=0$ )
ПБ	—	M	—	
П	ЕД	—	Y	$y := 1$ (так как $x > 0$ )
M: СТОП	—	—	—	

Здесь ЕД — символическое обозначение ячейки, содержащей константу, равную 1.

Дадим некоторые комментарии к программе. Первая команда вырабатывает признак  $\omega = 0$ , если  $x \geq 0$ , и  $\omega = 1$ , если  $x < 0$ . Предположим, что исходное значение  $x$  отрицательно. Тогда  $\omega = 1$  и команда с кодом П0 определит для исполнения следующую за ней команду:

В — ЕД Y,

по которой из нуля вычитается единица, и результат зашлется в ячейку Y. Вслед за этим произойдет безусловный переход на команду с меткой M. Итак, при  $x < 0$  вычислилось значение  $y = -1$ , что и требовалось в этом случае.

Предположим теперь, что исходное значение  $x$  было неотрицательным. Тогда первая команда программы

В X —

выработает признак  $\omega = 0$  и команда с кодом П0 передаст управление на команду с меткой L:

L: В — X —

Эта команда нам понадобится для того, чтобы выяснить, каково значение  $x$ : равно нулю или строго положительно. Если  $x = 0$ , то вырабатывается  $\omega = 0$ , если  $x > 0$ , то  $\omega = 1$ . Пусть в нашем случае  $x > 0$ , тогда значение  $(-x)$  отрицательно,  $\omega = 1$  и команда

П1 — M — 1 —

передаст управление команде

П ЕД — Y

В ячейку Y зашлется единица, что и нужно в случае  $x > 0$ . Наконец, если  $x = 0$ , то выполнится команда, следующая за командой с кодом П1:

П — — Y

в результате чего в ячейку Y зашлется нуль, а затем команда безусловной передачи управления определит в качестве очередной для исполнения команду СТОП.

Заметим, что в программе использовались все рассмотренные нами виды команд передачи управления: П0, П1 и ПБ.

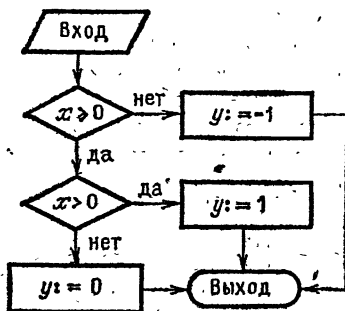


Рис. 4.8



Ту же задачу можно решить и при помощи другой программы:

П	ЕД	—	Y	$y := 1$
В	—	X	—	$-x \geq 0?$ (если $x \leq 0$ , то $\omega = 0$ ; если $x > 0$ , то $\omega = 1$ )
П1	—	L	—	
В	X	—	—	$x \geq 0?$ (если $x = 0$ , то $\omega = 0$ ; если $x < 0$ , то $\omega = 1$ )
П0	—	L-1	—	
В	—	ЕД	Y	$y := -1$
ПВ	—	L	—	
П	—	—	Y	$y := 0$
L: СТОП	—	—	—	

Нетрудно заметить, что в обеих программах для выработки признака  $\omega$  один раз было удобнее из  $x$  вычесть нуль, а в другой раз наоборот — из нуля вычесть  $x$ .

Во втором варианте программы мы прежде, чем выяснить знак числа  $x$ , сразу же выполнили присваивание  $y := 1$ , независимо от исходного значения  $x$ . Лишь затем мы выяснили  $x > 0$  или  $x \leq 0$ . Команда

В — X —

вырабатывает признак  $\omega = 1$ , если  $x > 0$ , и  $\omega = 0$ , если  $x \leq 0$  (так как по этой команде результат вычисления равен  $-x$  и  $\omega = 0$ , если  $-x \geq 0$ , т. е.  $x \leq 0$ ).

Если оказалось, что  $x > 0$ , то осуществляется переход на команду СТОП. При этом, как мы помним, в ячейке результата уже есть единица. Если же  $x \leq 0$ , то команда

В X —

выясняет  $x = 0$  или  $x < 0$  (вспомним, что в ячейке с нулевым номером хранится нуль). Если  $x = 0$ , то осуществляется переход на команду

П — — Y

по которой происходит присваивание  $y := 0$ .

Если же  $x < 0$ , то по команде

В — ЕД Y |  $y := 0 - 1$

в ячейку Y зашлется  $-1$ , после чего произойдет выход на команду СТОП.

Второй вариант программы оказался на одну команду короче.

Приведенный пример показал, что даже простейшие алгоритмы можно реализовать по-разному. Однако хотелось бы предостеречь от чрезмерного увлечения оптимизацией программ, т. е. от стремления сократить количество команд, рабочих ячеек, меток, констант и т. д. Как правило, программы при этом становятся логически более слож-

ными, и в них может быть больше ошибок, чем в более громоздких, но логически более простых и прозрачных программах. Да к тому же и ошибки в таких «оптимальных» программах искать значительно труднее.

## § 4.6. Программирование циклов

Команды перехода, с которыми мы познакомились выше, позволяют нам программировать и циклические процессы. В § 4.1 мы уже рассказывали о том, что бывают различные виды циклов — с заранее известным числом повторений цикла и с заранее неизвестным числом таких повторений.

Рассмотрим сначала программирование циклов с заранее неизвестным числом повторений. Легче всего понять принципы составления таких программ на простых примерах. Мы уже излагали ранее алгоритм нахождения остатка  $r$  от деления  $a$  на  $b$  — двух целых неотрицательных чисел ( $b \neq 0$ ) (рис. 4.9).

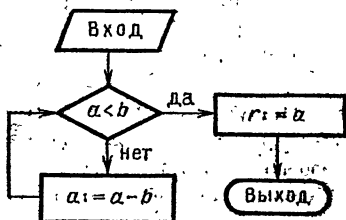


Рис. 4.9

Программа, реализующая этот алгоритм, выглядит так:

М:	В	А	В	—	$a < b?$
П1	—	Н	—	—	
В	А	В	А	—	$a := a - b$
ПВ	—	М	—	—	
Н:	П	А	—	Р	$r := a$
СТОП	—	—	—	—	

В команде с меткой М выясняется, что меньше: значение переменной  $a$  или значение переменной  $b$ , и если  $a < b$ , то вырабатывается признак  $\omega = 1$  и команда

П1 — Н —

осуществляет переход на команду с меткой N. Здесь произойдет результирующая пересылка, т. е. остаток от деления  $a$  на  $b$  занесется в ячейку с меткой R. Вслед за этим идет команда СТОП. Если же после первой команды результат получится неотрицательным, т. е.  $a \geq b$ , то выработается признак  $\omega = 0$ , и вслед за командой условной передачи управления выполнится команда

В А В А

В результате ее выполнения произойдет присваивание переменной  $a$  нового значения:  $a := a - b$ . И следом выполнится команда безусловного перехода

ПВ — М —

Теперь уже новое значение  $a$ , равное разности предыдущего значения  $a$  и  $b$ , сравнится с  $b$ . И опять, если это новое значение  $a$  будет меньше  $b$ , то произойдет переход к команде с меткой  $N$ , которая осуществит результирующую пересылку. В случае, если опять  $a \geq b$ , снова произойдет возврат к команде с меткой  $M$ , и уменьшение  $a$  ( $a := a - b$ ), и т. д.

Выполнение команды, в которой происходит присваивание  $a := a - b$ , будет повторяться до тех пор, пока  $a$  не станет меньше  $b$ . Очевидно, это произойдет обязательно, но заранее неизвестно, сколько раз придется повторить присваивание  $a := a - b$ .

Приведем еще один пример циклического процесса, когда число повторений цикла заранее не определено. Как нам известно из главы II, для вычисления значения функции  $y = e^x$  пользуются разложением этой функции в бесконечную сумму:

$$y = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots$$

(напомним, что  $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$ ).

Мы будем суммировать слагаемые до тех пор, пока очередное слагаемое по абсолютной величине не станет меньше некоторого наперед заданного малого положительного числа  $\varepsilon$  (например,  $\varepsilon = 0,001$ ). Такой момент обязательно наступит, так как  $n!$  растет быстрее, чем  $x^n$  для любого  $x$ . Получившуюся сумму можно принять за приближенное значение  $e^x$ .

Блок-схема алгоритмического процесса решения задачи представлена на рис. 4.10.

В первом блоке получают начальные значения переменных величины:  $y$  — результат,  $a$  — очередное слагаемое,  $i$  — целочисленная переменная, которая нужна для вычисления значения очередного слагаемого  $a$ . Следующий блок ( $y := y + a$ ) производит суммирование текущего значения  $y$  с очередным слагаемым  $a$ . После выполнения этого

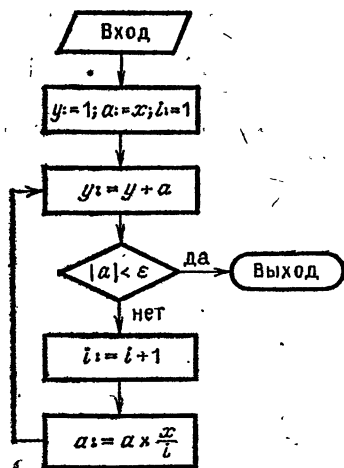


Рис. 4.10

блока  $y$  будет равно  $x + 1$ . Затем абсолютная величина  $a$  (в данном случае  $a$  равно  $x$ ) сравнится с  $\varepsilon$ . Сравнить нужно именно абсолютную величину  $a$  с  $\varepsilon$ , ибо, если бы мы написали просто  $a < \varepsilon$ , то для любого отрицательного  $x$  условие сразу же оказалось бы выполненным. Например, при  $x = -10$  значение  $a$  тоже будет равно  $(-10)$  и

$(-10) < 0,001$ , т. е.  $a < \varepsilon$  и мы сразу вышли бы из цикла, что, очевидно, преждевременно.

Если условие  $|a| < \varepsilon$  окажется выполненным, то это значит, что добавленное к  $y$  слагаемое уже достаточно мало, и вычислительный процесс можно завершить, приняв  $y$  за результат. Если же окажется, что  $|a| \geq \varepsilon$ , то перейдем на блоки, где выполняются действия:  $i := i + 1$  и  $a := a \times x / i$ . При первом их выполнении  $i$  получит значение, равное 2, а значение  $a$  будет равно  $x^2/2$  (так как старое значение  $a = x$ ). После этого произойдет безусловный переход на блок, где это  $a$  добавится к  $y$ :  $y := y + a$ . Старое значение  $y$  было равно  $1 + x$ , и к нему добавилось  $a = x^2/2$ . Теперь  $y$  получил значение

$$1 + x + x^2/2.$$

Затем новое значение  $a$  (равное  $x^3/2$ ) сравнится по абсолютной величине с  $\varepsilon$  и, если будет  $|a| \geq \varepsilon$ , то снова выполнятся действия  $i := i + 1$  и  $a := a \times x / i$ . Теперь уже  $i$  и  $a$  получат соответственно значения  $i = 3$  и  $a = x^3/3!$ , так как предыдущие значения были  $i = 2$  и  $a = x^2/2$ . Снова выполнится блок, где  $y := y + a$ . Теперь уже  $y$  будет равен

$$1 + x + x^2/2! + x^3/3! \quad (2! = 1 \cdot 2; 3! = 1 \cdot 2 \cdot 3).$$

Снова будет сравнение абсолютной величины очередного слагаемого  $x^3/3!$  с  $\varepsilon$  и т. д. До тех пор, пока, наконец, не выполнится условие  $|a| < \varepsilon$ .

Очевидно, количество повторений цикла зависит от  $x$  и от  $\varepsilon$  — чем больше значение абсолютной величины  $x$  и меньше  $\varepsilon$ , тем большее количество раз придется повторить шаги циклического процесса для получения результата с заданной точностью.

Запишем предложенный алгоритм в виде программы

П	ЕД	—	Y	$y := 1$
П	X	—	A	$a := x$
П	ЕД	—	I	$i := 1$
ЦИКЛ:	C	Y	A	$y := a + y$
	BA	A	EPS	$ a  < \varepsilon?$
	П	—	L	—
	C	I	ЕД	$i := i + 1$
	Y	A	X	$a := a \times x$
	D	A	I	$a := a / i$
	ПБ	—	ЦИКЛ	—
L:	СТОП	—	—	—

Легко убедиться в том, что эта программа в точности соответствует описанной выше блок-схеме алгоритма. Первые три команды и последняя выполняются по одному разу, а остальные многократно, причем число повторений циклического процесса заранее неизвестно и зависит от значения  $x$  и заданной точности вычислений  $\varepsilon$ .

Теперь мы рассмотрим программирование циклов с заранее известным числом повторений. Пусть нам надо составить программу суммирования 1000 чисел, расположенных подряд в ячейках памяти машины. Если мы для этих целей попытаемся воспользоваться только теми командами, которые рассмотрены ранее, то нам потребуется написать программу, состоящую из 1000 команд, следующего вида:

C B A+1	B		b:=b+a <sub>1</sub>
C B A+2	B		b:=b+a <sub>2</sub>
C B A+3	B		b:=b+a <sub>3</sub>
C B A+4	B		b:=b+a <sub>4</sub>
...	...		...
C B A+1000	B		b:=b+a <sub>1000</sub>

Здесь B — ячейка для результата, A+1 — адрес ячейки, где расположено первое слагаемое, A+2 — адрес следующей ячейки, где расположено второе слагаемое, в A+3 — третье слагаемое и т. д.

Легко заметить, что такая непомерно длинная программа состоит из однотипных команд, отличающихся друг от друга только тем, что в каждой следующей команде второй адрес увеличивается на единицу по сравнению с предыдущей.

Если бы в нашем распоряжении была команда, которая при выполнении автоматически наращивала свой второй адрес путем прибавления к нему 1 и при повторном к ней обращении выполняла бы то же арифметическое действие, но уже с новым адресом, увеличенным на 1, то можно было бы предыдущую программу суммирования записать более компактно — всего в несколько строк.

В самых первых ЭВМ проблема изменения адресов команды в цикле работы машины решалась следующим образом. Команды программы располагаются в ячейках памяти наравне с числами и так же, как и числа, представляют собой набор нулей и единиц, т. е. некоторое двоичное слово. Следовательно, можно, пользуясь командами машины, изменять содержимое ячейки, в которой расположена команда.

Можно подобрать такую константу, прибавление которой к команде изменит величину закодированного в ней адреса на заданную величину. Затем можно выполнить эту измененную, модифицированную команду. Выполняя в цикле модификацию команды в памяти и выполнение измененной команды, можно было создать достаточно компактные циклические программы. В случае задачи суммирования блок-схема такого программного цикла могла бы выглядеть, как на рис. 4.11. В этом случае на каждое полезное действие накопления искомой суммы выполняется еще и действие по модификации команды. Такая программа будет выполняться вдвое медленнее по сравнению с программой, в которой все действия вытянуты в линию.

Модификация программы в ходе вычислений приводит и к другим неприятностям. Например, выполненную программу нельзя повто-

ритель — надо восстановить исходный вид ее команд, «испорченных» модификацией.

Естественно, разработчики ЭВМ не могли с этим примириться. Были предложены и аппаратно реализованы такие средства, которые позволяют, не меняя сами команды в памяти, тем не менее модифицировать (изменять) адреса, по которым извлекаются операнды из памяти.

Идея всех этих реализаций изменения адресов состоит в следующем. Код, записанный в поле адреса команды, рассматривается как неизменяемая часть адреса. Для того чтобы получить реальный адрес, к этому коду следует добавить некоторое число, находящееся в специально выделенном для этих целей регистре (или регистрах) машины.

Таким образом вычисленный адрес в команду не заносится и ее не меняет. Вычисленный адрес — отличие от кода адреса в команде называют *исполнительным* адресом. Этот исполнительный адрес и является тем самым адресом, с которым в действительности выполняется машинная команда. Регистр, в котором хранится добавка, называется *регистром адреса*. Если в регистре адреса находится число, равное нулю, то к коду команды ничего не добавляется и сам код адреса становится исполнительным адресом.

В машинах существуют специальные команды, с помощью которых можно менять содержимое регистра адреса. Такие команды называются командами *адресной арифметики*. Эти команды модификации кода адреса выполняются очень быстро и не слишком замедляют выполнение программы.

В современных машинах регистров адреса обычно несколько (8, 16 иногда 32), это позволяет запастись разными «добавками» и использовать их по мере надобности. При наличии нескольких регистров адреса в поле адреса машинной команды, кроме кода адреса (постоянной части) следует также указывать, какой регистр адреса используется для получения исполнительного адреса.

В общем случае адресное поле команды содержит информацию о способе вычисления исполнительного адреса. В современных ЭВМ различных способов получения исполнительного адреса насчитывается около 20. Рассмотренный выше способ суммирования кода с добавком

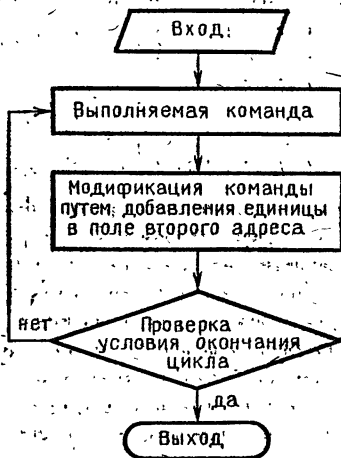


Рис. 4.11

(приращением), взятым из регистра адреса, является самым простым и в то же время самым употребительным.

Мы будем использовать простейший способ модификации команд, когда в машине есть всего один регистр адреса (РА).

В этом случае каждый адрес в любой команде распадается на две составляющие — постоянную и переменную. Постоянная составляющая пишется непосредственно в адресной части команды. Именно эти адреса мы и записывали до сих пор в командах наших программ. Но кроме постоянных составляющих есть еще и переменные части адресных полей команд. Они-то и записываются в регистре адреса. Переменная часть может добавляться или не добавляться к постоянной составляющей адреса. Этим управляют в каждой программе признаки модификации. Происходит это следующим образом. В каждой ячейке три определенных двоичных разряда отводятся под признаки модификации, соответствующие каждому своему адресу в команде. Если некоторый адрес в команде нужно модифицировать, то в соответствующий разряд признака модификации заносят единицу, если же адрес не нужно модифицировать, то в соответствующий признак модификации заносят нуль.

Мы здесь не будем останавливаться на том, как это делается при написании программ в машинных кодах.

Покажем, как осуществляется модификация адресов в командах, написанных на автокоде.

Для того чтобы модифицировать какой-либо адрес, т. е. добавить к этому адресу содержимое регистра адреса, достаточно в записи команды в поле данного адреса справа от модифицируемого адреса, поставить круглые скобки.

Пусть, например, в регистре адреса находится число 1 и выполняется команда

$C\ B\ A0\ B$

Тогда произойдет сложение содержимого ячеек  $B$  и  $A+1$ , результат сложения зашлется в ячейку  $B$ .

Еще пример. Команда, с помощью которой мы хотим выполнить присваивание  $z_i := x_i + y$  (индекс  $i$  говорит о том, что берутся  $i$ -е компоненты массивов) выглядит так:

$C\ X0\ Y\ Z0$  (1)

В команде модифицируются первый и третий адрес. Если к моменту выполнения этой команды в регистре адреса находится 0, то исполнительные адреса совпадают с теми, что записаны в команде. Если же в регистре стояла единица, то она добавится к 1-му и 3-му адресам, и хотя внешний вид самой команды не изменится, выполнится она так, как если бы имела вид

$C\ X+1\ Y\ Z+1$

Таким образом, сложится уже содержимое ячеек  $X+1$  и  $Y$ , а результат зашлется в ячейку  $Z+1$ . Подчеркнем еще раз, что в таком виде команду нигде писать не нужно, она имеет вид (1), а исполнительный вид команды после ее модификации формируется в устройстве управления, и в таком сформированном виде команда выполняется машиной.

Если теперь нужно сложить с содержимым ячейки  $Y$  содержимое ячейки  $X+5$  и результат заслать в ячейку  $Z+5$ , то достаточно выполнить все ту же команду (1), но только к моменту ее выполнения в регистре адреса теперь уже должно находиться целое число 5.

Модифицировать можно как каждый адрес в отдельности, так и одновременно два или три адреса.

Но так как в нашей машине лишь один регистр адреса, то ко всем адресам в команде может добавляться лишь одно и то же значение, хранящееся в этом регистре адреса. Так в команде (1) мы прибавляем к 1-му и 3-му адресам одновременно то 1, то 5. Безусловно это создает определенное неудобство, ибо не позволяет один адрес в команде изменять на одно значение, а другой — на другое.

Итак, для того чтобы менять номера ячеек, участвующих в команде, вовсе не требуется менять вид самой команды, достаточно лишь к моменту очередного ее выполнения соответствующим образом сформировать содержимое регистра адреса.

Команда, формирующая новое значение регистра адреса, имеет вид

$$PA - n - | pa: = n \quad (2)$$

Эта команда засылает в регистр адреса код, записанный в поле второго адреса

$$pa: = n \quad (pa - \text{содержимое регистра адреса}).$$

Заметим, что, в отличие от всех рассмотренных ранее команд, в команде с кодом операции  $PA$  происходит засылка непосредственно самого исполнительного адреса в регистр адреса (а не содержимого ячейки с этим адресом)

Например, команда

$$PA - 1 -$$

зашлет в регистр адреса единицу ( $pa: = 1$ ), а команда

$$PA - 5 -$$

зашлет в регистр адреса число 5 ( $pa: = 5$ ). Нетрудно видеть, что с помощью той же команды с кодом операции  $PA$  можно к содержимому регистра адреса добавлять какое-либо число. Пусть, например, содержимое регистра адреса надо увеличить на 1. Для этого достаточно выполнить команду

$$PA - 10 - \quad (3)$$

Тогда второй исполнительный адрес этой команды будет модифицирован, т. е. к адресу, записанному в самой команде (в нашем случае



это единица) добавится значение  $pa$ , находящееся в регистре адреса к моменту выполнения команды.

Тем самым исполнительный 2-й адрес станет равным  $pa + 1$ , и по команде (3) это значение зашлется в регистр адреса. Если бы нам надо было увеличить содержимое регистра адреса на 3, мы бы выполнили команду

РА — 30 —

Таким образом, у нас есть средство, позволяющее менять исполнительные адреса в командах, без изменения самих этих команд, если использовать модификацию этих адресов с помощью регистра адреса; в нужный момент меняя его содержимое.

Регистр адреса может оказаться исключительно полезным и при организации циклических процессов. Им, в частности, можно воспользоваться для проверок на окончание цикла. В этом регистре в ходе выполнения цикла шаг за шагом накапливается приращение, модифицирующее исполнительный адрес. Если в программе на каждом шаге цикла к содержимому регистра адреса прибавляется единица, то тем самым он может рассматриваться как счетчик числа повторений, одновременно используемый для модификации адресов.

Кроме условных передач управления по значению признака  $o$ , характеризующего результат выполнения арифметических операций, в ЭВМ используются условные переходы по значению  $pa$ .

Чтобы осуществить проверку по счетчику циклов, в качестве которого выступает  $pa$ , необходимо проверить достиг ли  $pa$  того значения, после которого надо выйти из цикла.

Если предельное значение числа повторений цикла мы обозначим буквой  $N$ , то эта операция перехода может быть сформулирована так:

если  $pa < N$ , то следует повторить цикл,

если  $pa \geq N$ , то следует завершить цикл.

Эти действия, т. е. изменение значения регистра адреса и сравнение его с конечным значением, в машинах часто объединяются в одной команде.

Такую команду на языке автокода мы обозначим ПМ (Переход при значении регистра адреса меньше  $N$ ):

ПМ A1 A2 A3

И в этой команде, как и в других, каждый адрес может модифицироваться. Поэтому мы будем говорить об исполнительных адресах команды. Смысл команды состоит в следующем. Происходит сравнение текущего значения содержимого регистра адреса с первым исполнительным адресом (именно адресом, а не содержимым ячейки с этим адресом). Если  $pa < A1$  (исп), то осуществляется переход по адресу  $A2$  (исп); если же  $pa \geq A1$  (исп), то сохраняется естественный порядок выполнения команд, т. е. выполняется команда из сле-

дующей по порядку ячейки. Кроме того, в регистр адреса зашлется новое значение  $A3$  (исп).

Пусть, например, в регистр адреса содержится слово  $(0003)_8$  и выполняется команда:

К: ПМ 4 N 10

В результате выполнения этой команды текущее значение регистра адреса, а именно  $(0003)_8$  сравнится с  $A1$  (исп)  $= (4)_{10} = (0004)_8$  и, так как условие  $3 < 4$  соблюдено, то произойдет переход к команде с меткой  $N$ . Кроме того, в регистр адреса зашлется новое значение  $ra := A3$  (исп):

$ra := ra + 1$ .

В нашем случае текущее значение  $ra$  равно 3. Оно увеличится на 1, получится 4 и зашлется в регистр адреса. Отметим, что исполнительные адреса формировались с использованием старого значения регистра адреса; оно же сравнивалось с  $A1$  (исп), и лишь затем в адресный регистр засылается новое значение. Причем это новое значение формируется независимо от результата сравнения.

Если затем придется снова выполнять команду с меткой  $K$  уже с новым значением содержимого регистра адреса ( $ra = 4$ ), то теперь при сравнении его с  $A1$  (исп) получится, что  $A1$  (исп)  $= ra$  ( $4 = 4$ ) и, поэтому, следующей будет выполняться команда из ячейки  $K+1$  (условие  $ra < A1$  (исп) оказалось невыполненным). Кроме того, в регистр адреса зашлется значение:

$ra := ra + 1 = 4 + 1 = 5$

Теперь мы уже можем программировать циклические процессы с использованием регистра адреса. Составим программу для вычисления  $y = ax^6$ . Блок-схема алгоритма приведена на рис. 4.12.

Программа на языке символического кодирования:

П	A	—	Y	$y := a$
РА	—	1	—	$ra := 1$
М: У	Y	X	Y	$y := y \times x$
ПМ	6	M	10	при $ra < 6$ переход на М; $ra := ra + 1$
СТОП	—	—	—	

Как видно из программы, здесь регистр адреса используется только для управления числом повторений цикла. Фактически он играет ту же роль, что и переменная  $i$  в блок-схеме.

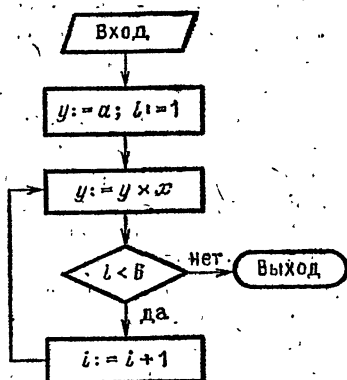


Рис. 4.12

Сначала выполняется засылка в регистр адреса единицы ( $pa := 1$ ), затем происходит операция умножения  $y := y \times x$ . В результате первого выполнения этого умножения получим:  $y = ax$ . Затем текущее значение адресного регистра, т. е. единица сравнится с числом 6. Так как 1 меньше, чем 6, то произойдет переход на метку М. Кроме

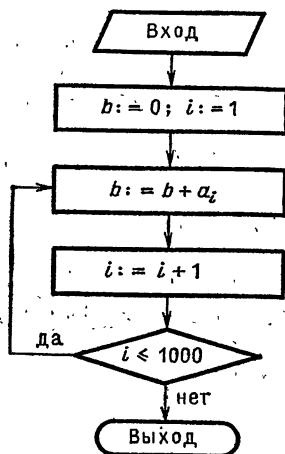


Рис. 4.13

того вычислится новое значение  $pa := pa + 1$ , т. е. будет  $pa = 2$ . Здесь снова выполнится  $y := y \times x$ . Но к этому моменту  $y$  имел значение  $ax$ . Поэтому получится новое значение  $y = ax^2$ . Текущее значение  $pa$  (теперь оно равно 2) сравнится с числом 6. Опять выполнится условие  $pa < A1$  (исп) (так как  $2 < 6$ ), снова увеличится значение адресного регистра  $pa := pa + 1$ , т. е. теперь  $pa = 3$ , и снова произойдет переход по метке М. Вычислится  $y = ax^3$  и т. д. Наконец, к моменту, когда вычислится  $y = ax^6$ , значение содержимого адресного регистра будет равно 6. Теперь уже условие  $pa < A1$  (исп) не будет выполняться (так как  $6 = 6$ ) и, следовательно, выполнится команда из следующей ячейки (СТОП). Кроме того, как и прежде, изменится значение

адресного регистра:  $pa := pa + 1$ , в результате чего после завершения выполнения программы  $pa$  будет равно 7.

Еще раз подчеркнем, что регистр адреса в рассмотренном примере использовался только как счетчик количества повторений цикла. Теперь приведем пример программы, в которой регистр адреса будет использоваться одновременно и для подсчета количества повторений цикла и для формирования переменных команд.

Вернемся к нашей задаче суммирования 1000 чисел:

$$b = a_1 + a_2 + a_3 + \dots + a_{1000}.$$

Блок-схема алгоритма суммирования представлена на рис. 4.13.

Тот же алгоритм на языке символического кодирования:

П	—	→	B	$b := 0$
РА	—	1	—	$pa := 1 \ (i := 1)$
L: С	B	A0	B	$b := b + a_i$
ПМ	1000	L	10	при $pa < 1000$ переход на L; $pa := pa + 1 \ (i := i + 1)$
СТОП	—	—	—	

Здесь все управление циклическим процессом сосредоточено в одной команде:

ПМ 1000 L 10

(напомним, что ПМ означает переход при значении регистра адреса  $ra$  меньше, чем  $A1$  (исп)).

С одной стороны, эта команда меняет содержимое регистра адреса, каждый раз увеличивая его на единицу, и тем самым модифицирует переменный адрес в команде с меткой  $L$ . С другой стороны, сравнивает значение регистра адреса с конечным значением, в данном случае с числом 1000. Таким образом, происходит управление как переменными адресами, так и числом повторений цикла. Поэтому вычислительный процесс здесь протекает следующим образом: сначала переменные величины  $b$  и  $i$  принимают начальные значения  $b=0$  и  $ra=1$  (содержимое адресного регистра играет роль номера  $i$ , или счетчика компонент суммируемого массива), а затем начинается сам вычислительный процесс.

При первом выполнении команды с меткой  $L$  произойдет операция присваивания  $b := b + a_i$ . В команде с кодом ПМ текущее значение адресного регистра, т. е.  $ra=1$  сравнится с числом 1000. Так как  $1 < 1000$ , то произойдет переход по метке  $L$ . При этом значение адресного регистра увеличится на единицу (третий адрес команды модифицируется)  $ra := ra + 1$ , т. е. получится  $ra=2$ . Снова выпол-

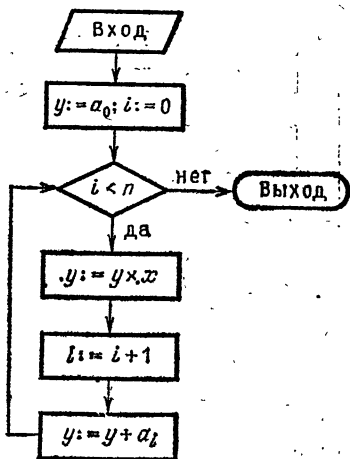


Рис. 4.14

нится команда с меткой  $L$ , но теперь уже с новым значением адресного регистра. Поэтому к полученному ранее значению  $b$  добавится уже  $a_2$ , т. е.  $b := b + a_2$  и теперь значение  $b$  будет равно сумме  $a_1 + a_2$ . Опять выполнится команда с кодом ПМ. Значение адресного регистра, теперь уже равное 2, сравнится с числом 1000 и т. д. Все это будет продолжаться до тех пор, пока, наконец, выполнится присваивание:  $b := b + a_{1000}$ , а значит мы получим искомую сумму. В команде с кодом ПМ значение  $ra$ , равное 1000, сравнится с числом 1000 и так как 1000 не меньше, чем 1000, то произойдет выход из цикла и переход к следующей по порядку команде — команде СТОП. Заметим, что при этом значение адресного регистра станет равным числу 1001.

Приведем еще пример алгоритма с известным числом повторений цикла. Вычислим значение многочлена:

$$P_n(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \dots + a_{n-1}x + a_n$$

для некоторого значения переменной  $x$ . Воспользуемся для этого

представлением полинома по схеме Горнера:

$$P_n(x) = a_0 x^n + a_1 x^{n-1} + \dots + a_{n-1} x + a_n = \\ = (\dots ((a_0 x + a_1) x + a_2) x + \dots + a_{n-1}) x + a_n.$$

Блок-схема алгоритма представлена на рис. 4.14.

Составим программу на языке символьного кодирования. Пусть коэффициенты полинома находятся в подряд расположенных ячейках, начиная с  $A$ , т. е.  $a_0$  в ячейке  $A$ ,  $a_1$  в ячейке  $A+1$  и т. д., и пусть степень многочлена  $n=50$ :

П	A	—	Y		$y := a_0$
РА	—	1	—		$ra := 1 \ (i := 1)$
K: Y	Y	X	Y		$y := y \times x$
С	Y	A0	Y		$y := y + a_i$
ПМ	50	K	10		при $ra < 50$ переход по метке K;
СТОП	—	—	—		$ra := ra + 1 \ (i := i + 1)$

Внимательный читатель заметил, что последовательность действий в блок-схеме и программе отличаются друг от друга и что блок-схема алгоритма позволяет вычислять значение полинома любой степени, в том числе и нулевой. В то же время написанная выше программа при замене в последней команде числа 50 на соответствующее  $n$  будет верно работать только при  $n \geq 1$ .

## § 4.7. Программирование вложенных циклов

Рассмотрим теперь программирование вложенных циклов. Речь идет об алгоритме с циклическими процессами, внутри которых есть тоже циклические процессы.

Пусть, например, надо вычислить таблицу значений функции  $y = e^x$  для различных значений аргумента:  $x_1, x_2, \dots, x_{50}$ . Блок-схему алгоритма и программу на языке символьного кодирования для вычисления одного значения функции  $y = e^x$  мы уже составляли (см. с. 147). Теперь представим блок-схему алгоритма для построения таблицы значений

$$y_i = e^{x_i} \quad (i = 1, 2, \dots, 50)$$

(рис. 4.15).

Здесь имеется два цикла — один для вычисления функции  $y = e^x$  с заданной точностью  $\varepsilon$  для некоторого конкретного значения аргумента  $x$ , второй — для организации многократной работы первого цикла при различных значениях аргумента. Первый цикл на блок-схеме обозначен пунктиром. Он является частью второго цикла и поэтому называется внутренним циклом, а второй — внешним. Во внутреннем

цикле аргумент функции всегда один и тот же — переменная  $v$ , а результат — переменная  $u$ . Внешний цикл обеспечивает, чтобы переменной  $v$  присваивались поочередно новые значения аргумента  $x_i$  и чтобы получившиеся результаты также последовательно присваивались очередным  $y_i$ .

Внутренний цикл заканчивается при достижении заданной точности в вычислении одного значения функции  $y = e^x$ , внешний цикл заканчивается, если значение функции вычислено для всех значений аргумента. Число повторений внутреннего цикла заранее неизвестно и определяется значениями  $x_i$  и  $\varepsilon$ , а количество повторений внешнего цикла задано.

Представим теперь алгоритм в виде программы на языке символического кодирования, предположив, что аргументы  $x_i$  находятся в ячейках с адресами  $X+i$ , а результаты вычислений надо поместить соответственно в ячейки  $Y+i$  ( $i=1, 2, \dots, 50$ ):

РА	—	1	—
К: П	X0	—	V
П	ЕД	—	U
П	ЕД	—	J
П	V	—	A

M:	C	U	A	U
	BA	A	EPS	—
	П1	—	N	—
	C	J	ЕД	J
	У	A	V	A
	Д	A	J	A
	ПБ	—	M	—

N: П	U	—	Y0
ПМ	50	K	10

СТОП — — —

ра:= 1 ( $i:=1$ )

$v:=x_i$

$u:=1$

$j:=1$

$a:=v$

$u:=u+a$

$|a| < \varepsilon$

$j:=j+1$

$a:=a \times v$

$a:=a/j$

$y_i:=u$

при  $pa < 50$  переход по метке K;

$pa:=pa+1$  ( $i:=i+1$ )

В программе внутренний цикл обведен рамкой. По существу он ничем не отличается от программы для вычисления функции  $y = e^x$ , приведенной на с. 148. Во внешнем цикле в команде с меткой К происходит засылка очередного аргумента  $x_i$  из ячейки  $X+i$  в рабочую ячейку V, а в команде с меткой N пересылается результат в очередную ячейку  $Y+i$ .

Тем самым в ячейке  $Y+i$  оказывается результат вычисления значения функции

$$y_i = e^{x_i}.$$

Регистр адреса играет роль счетчика количества повторений внутреннего цикла и используется для модификации адресов, соответствующих  $x_i$  и  $y_i$ . Изменением значения адресного регистра и количеством повторений внутреннего цикла управляет команда с кодом ПМ. Если бы понадобилось получить таблицу не для

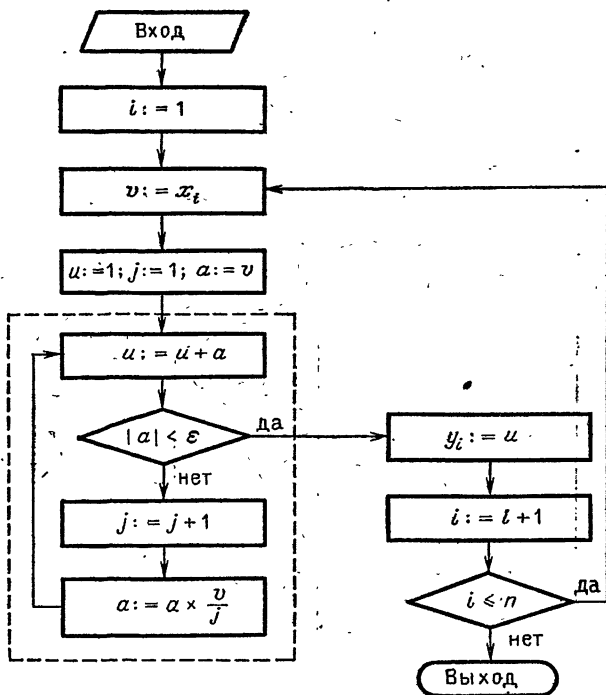


Рис. 4.15

50 значений аргумента, а для какого-то другого числа значений, то надо было бы изменить 1-й адрес в команде с кодом ПМ.

В нашей программе адресный регистр используется только во внешнем цикле.

## § 4.8. Подпрограммы

Представим себе, что при записи алгоритма решения некоторой задачи мы столкнулись с ситуацией, когда одна и та же последовательность действий должна выполняться во многих различных местах алгоритма. Конечно, можно было бы в каждом таком месте вставлять одинаковые последовательности команд, реализующие эти действия.

Программа от этого бы разрослась, да и сам процесс многократного написания такой последовательности команд лишь один раз и каким-то простым способом обращаться к ней из разных мест программы, а затем после ее выполнения возвращаться в то место программы, откуда произошло обращение.

Такую выделенную последовательность команд, к которой можно обращаться из разных мест основной программы, называют *подпрограммой*.

Схематически обращение к подпрограмме можно изобразить следующим образом (см. рис. 4.16). Стрелками на схеме показан ход выполнения программы и подпрограммы. Цифры над стрелками указывают порядковый номер обращения к подпрограмме.

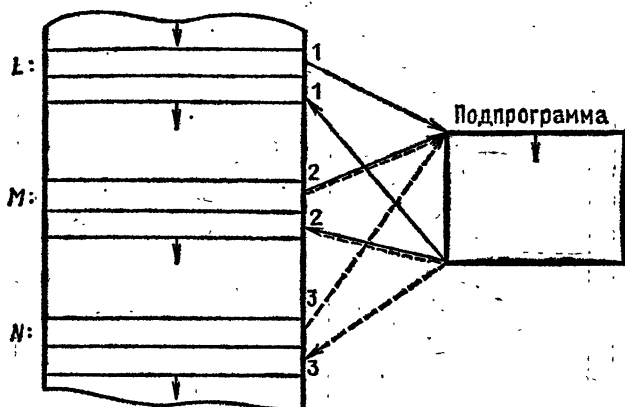


Рис. 4.16

Естественно считать, что результат работы подпрограммы зависит от каких-то исходных величин, задаваемых при обращении к подпрограмме. Такие величины обычно называют *параметрами* подпрограммы.

Что же нужно сделать, чтобы обратиться к подпрограмме? Во-первых, нужно задать ее параметры; во-вторых, нужно перейти на начало подпрограммы; в-третьих, нужно вернуться в основную программу по окончании работы подпрограммы. Легче всего решить второй вопрос — перейти на начало подпрограммы. Для этого достаточно знать метку первой команды подпрограммы и воспользоваться командой перехода по этой метке. Так, если подпрограмма начинается с ячейки *N*, то для обращения к подпрограмме достаточно в основной программе выполнить команду

ПБ — *N* —



В состоянии мы решить вопрос и о задании параметров. По существу нечто подобное мы делали в предыдущем разделе при программировании вложенных циклов. Там во внутреннем цикле аргумент всегда брали из одной и той же рабочей ячейки ( $V$ ) и результат тоже помещали всегда в одну и ту же рабочую ячейку ( $U$ ), а во внешнем цикле, прежде чем приступить к выполнению внутреннего цикла, в рабочую ячейку для аргумента, засылалось то фактическое значение, для которого нужно было вычислить результат. По окончании работы внутреннего цикла результат, полученный в рабочей ячейке, пересылали в отведенную для него ячейку памяти. Вот и в подпрограмме должны быть некоторые рабочие ячейки как для исходных данных, так и для результата. Перед каждым обращением к подпрограмме исходные данные нужно заслать в рабочие ячейки подпрограммы, а затем обратиться к подпрограмме. Она сформирует в определенной рабочей ячейке результат (возможно, что результат будет размещаться в нескольких рабочих ячейках), и после возвращения в основную программу можно либо переслать результат в какую-либо ячейку или ячейки, либо использовать в основной программе непосредственно эти рабочие ячейки с результатом. Нужно только помнить, что при следующем обращении к подпрограмме в этой рабочей ячейке уже будет получено другое значение.

Теперь рассмотрим вопрос о том, как организовать возврат из подпрограммы в основную программу. Ясно, что последней командой подпрограммы должна быть команда перехода на какую-то команду в основной программе. Чаще всего, хотя и не всегда, нужно вернуться к команде, следующей за командой обращения к подпрограмме. Такую ситуацию мы и изобразили на схеме.

Можно поступить следующим образом. Запомним информацию о месте команды обращения из основной программы к подпрограмме с помощью регистра адреса. А именно — перед переходом к подпрограмме в основной программе выполним команду

$PA - L -$

где  $L$  — адрес команды обращения к подпрограмме.

Пусть вернуться из подпрограммы надо к следующей строке. Тогда последней командой подпрограммы будет безусловный переход:

$PB - 10 -$

(6)

Исполнительный адрес здесь будет  $L+1$  (так как  $pa=L$ ). Таким образом, возврат произойдет в ячейку  $L+1$ , т. е. следующую за той, где находится обращение. При следующем обращении к подпрограмме перед командой обращения надо не забыть выполнить команду

$PA - M -$

где  $M$  — адрес ячейки очередного обращения к подпрограмме.

В этом случае по окончании работы подпрограммы команда (6) осуществит безусловный переход на команду по адресу  $M+1$  (так как теперь  $pa = M$ ).

Так можно поступать при каждом обращении к подпрограмме. Но при таком способе обращения к подпрограмме могут возникнуть определенные затруднения. А именно, мы входим в подпрограмму со значением регистра адреса, которое нам понадобится в последней команде подпрограммы — команде возврата (6). И, стало быть, оно должно сохраниться к моменту выполнения этой команды. Но ведь в подпрограмме могут быть циклические процессы, которые потребуют использования регистра адреса. И тогда они «испортят» значение, необходимое для организации возврата. То есть нужно в начале работы подпрограммы запомнить это значение так, чтобы можно было его восстановить перед выполнением команды возврата.

В современных ЭВМ для организации обращения к подпрограммам с последующим возвратом в основную программу используются специальные дополнительные регистры возвратов, работающие по описанному выше принципу, и специальные команды перехода на подпрограмму.

Будем считать, что в нашей машине есть одна такая команда перехода и один регистр возврата.

Команда перехода на подпрограмму выполняет одновременно два действия: запоминает свой собственный адрес, увеличенный на 1, в регистре возвратов (PB) и осуществляет переход на начало подпрограммы.

Вид этой команды перехода с запоминанием возврата (ПВ) может быть таким:

$L: \text{ПВ} - N -$

где  $L$  — метка (адрес) самой команды перехода,  $\text{ПВ}$  — код операции перехода (Переход с Возвратом). Алгоритм работы команды таков:

1)  $pv := L + 1$

2) перейти к  $N$

( $pv$  — содержимое регистра возврата).

Команда возврата из подпрограммы выполняет переход по значению, сохраненному в регистре возвратов. Следовательно, в команде возврата, мнемонический код которой мы обозначим словом ВОЗВРАТ, нет необходимости указывать адреса, и она будет выглядеть в автокоде так:

$\text{ВОЗВРАТ} - - -$

Действие, выполняемое этой командой, эквивалентно выполнению команды

$\text{ПВ} - L + 1 -$

Таким образом, обращаться к подпрограмме можно с помощью обычного безусловного перехода (ПБ) на начало подпрограммы. Тогда в регистре адреса (РА) надо запоминать место команды перехода. А можно обратиться к подпрограмме и с помощью команды перехода с возвратом (ПВ).

В последнем случае место команды перехода автоматически запоминается в регистре возврата (РВ), и так как регистр адреса здесь уже не используется для организации возврата в основную программу, то мы можем им распоряжаться в подпрограмме по своему усмотрению.

Обращений к подпрограмме из разных мест основной программы может быть сколько угодно. При этом никаких изменений в подпрограмме делать не требуется — она выглядит всегда одинаково. Нужно лишь перед тем, как перейти к ней, заслат в рабочие ячейки текущие значения параметров подпрограммы.

Пусть, например, в основной программе нужно обратиться к подпрограмме, вычисляющей значение многочлена

$$P_{50}(x) = a_0 x^{50} + a_1 x^{49} + a_2 x^{48} + \dots + a_{49} x + a_{50}.$$

Программу для вычисления этого многочлена мы уже составляли (см. с. 156). Теперь оформим ее как подпрограмму:

$\bar{N}$ : РА	— 1 —	$pa := 1;$
П	A — Y	$y := a_0$
F: Y	Y X Y	$y := y \times x$
С	Y A() Y	$y := y + a_i$
ПМ	50 F 1()	при $pa < 50$ переход на F; $pa := pa + 1$
ВОЗВРАТ		возврат в основную программу

Здесь предполагается, что коэффициенты полинома  $a_0, a_1, \dots, a_{50}$  находятся соответственно в ячейках A, A+1, ..., A+50; X — рабочая ячейка для аргумента  $x$ , Y — рабочая ячейка для результата  $y$ .

Пусть теперь нам надо обратиться из разных мест основной программы к этой подпрограмме, чтобы вычислить первый раз значение

$$y_0 = P_{50}(x_0),$$

причем  $x_0$  находится в ячейке X0 и результат  $y_0$  должен быть помещен в ячейку Y0, а второй раз вычислить

$$y_1 = P_{50}(x_1),$$

$x_1$  — в ячейке X1,  $y_1$  нужно поместить в ячейку Y1.

Обращения к подпрограмме в основной программе будут выглядеть следующим образом:

П	X0 — X		$x := x_0$
L: ПВ	— N —		
П	Y — Y0		$y_0 := y$
...	...		
П	X1 — X		$x := x_1$
M: ПВ	— N —		
П	Y — Y1		$y_1 := y$

Сначала в рабочую ячейку X зашлется  $x_0$ . После этого произойдет переход к подпрограмме:

ПВ — N —

В регистр возврата при этом занесется  $L+1$ .

Затем вычислится значение многочлена. По команде возврата произойдет переход из подпрограммы в ячейку  $L+1$  основной программы. Там вычисленное значение перешлется в ячейку для результата Y0.

После этого в рабочую ячейку X зашлется новое значение  $x_1$  и снова осуществится переход к подпрограмме. Теперь уже после возврата из подпрограммы результат перешлется в ячейку Y1.

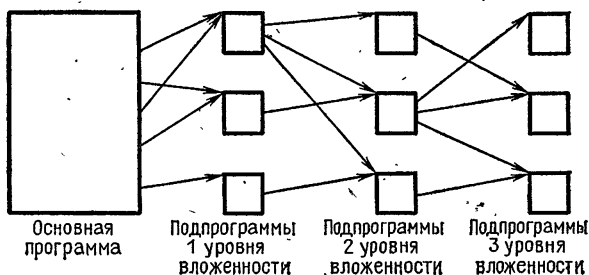
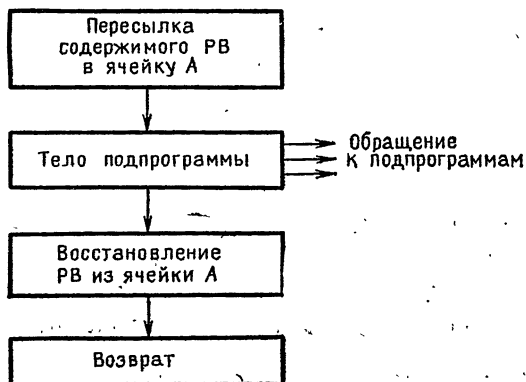


Рис. 4.17

При составлении программ часто бывает необходимо обращаться к подпрограммам, которые в свою очередь сами обращаются к подпрограммам. Возникает ситуация, которая называется вложенностью подпрограмм. Говорят об уровне вложенности, имея в виду число ступеней обращения из подпрограммы в подпрограмму. На схеме, приведенной на рис. 4.17, отмечены уровни вложенности.

В том случае, если в ЭВМ существует только один регистр возврата, для осуществления многоуровневого обращения из подпрограммы в подпрограмму приходится заботиться о том, чтобы сохранять содержимое регистра возврата при входе в подпрограмму, которая в свою очередь может обращаться к другой подпрограмме.

Перед выполнением команды ВОЗВРАТ из подпрограммы следует восстановить содержимое регистра возврата. Такие подпрограммы образуются по следующей схеме:



Для того чтобы упростить создание вложенных подпрограмм в современных ЭВМ, используется несколько регистров возврата, а в командах перехода на подпрограмму и в командах возврата надо указывать номер соответствующего регистра возврата.

Во многих ЭВМ, в частности в микро-ЭВМ, вместо регистров возврата используются ячейки оперативной памяти.

## ГЛАВА 5

# ПРОГРАММИРОВАНИЕ НА БЕЙСИКЕ

### § 5.1. Понятие о языках высокого уровня

В предыдущих разделах мы познакомились с некоторыми способами записи алгоритмов, в частности с представлением их в виде блок-схемы, словесно-формульной записи, на машинно-ориентированном языке в символьных обозначениях и в виде последовательности команд ЭВМ, т. е. машинном языке. Способ записи определяется целью, с которой мы записываем алгоритм,—если нам нужно представить себе принципиальную структуру алгоритма, его логические связи, то здесь удобна, например, блок-схема. Если же нам нужно, чтобы по выбранному алгоритму работала вычислительная машина, то в конечном итоге необходимо иметь программу на машинном языке.

Преимущества и недостатки указанных способов мы уже рассматривали ранее. Так, главное преимущество машинного языка заключается в том, что программа, написанная на таком языке, понимается электронной вычислительной машиной точно и однозначно, является для нее четким руководством к действию, в соответствии с которым ЭВМ вырабатывает вполне определенный и однозначный результат (результат—это не обязательно какое-либо число или даже набор чисел, это может быть любой текст, рисунок или даже какое-то действие, например обращение к другой программе). Но мы знаем и существенные недостатки машинного языка—написание программы на этом языке—чрезвычайно трудоемкий, утомительный процесс, сопровождающийся большим количеством ошибок. Человек, состав, ляющий такую программу, должен в совершенстве знать систему команд данной ЭВМ, организацию памяти и прочие ее особенности.

Заметим, кстати, что перечень различных операций, выполняемых современной ЭВМ, очень велик. Так, машины ЕС ЭВМ насчитывают более 200 видов машинных операций. Даже микро-ЭВМ умеют выполнять более 100 различных операций. Знать их все, уметь ими рационально пользоваться—задача непростая. Программа для решения любой более или менее серьезной задачи столь сложна и громоздка,

что ее не только другой человек, но и сам автор через небольшой промежуток времени уже не всегда в состоянии понять и в ней разобраться. Несколько улучшает дело пользование автокодом. Представление о нем дает машинно-ориентированный язык, о котором говорилось выше. Однако и здесь программа получается хотя и значительно более наглядной, но громоздкой, трудно обозримой, и тоже приходится учитывать особенности конкретной ЭВМ, в первую очередь систему команд.

В наше время многообразие типов вычислительных машин, а следовательно, и различных машинных языков столь велико, что это уже вызывает определенные трудности. Во многих организациях и предприятиях имеется по несколько ЭВМ различных типов, и программа, написанная на машинном языке одной ЭВМ, не может быть воспринята другой ЭВМ. В этом случае алгоритм должен быть заново написан на языке этой второй ЭВМ и отлажен. То есть обмен алгоритмами и программами, записанными на машинных языках, крайне ограничен. Он возможен только для машин, как говорят, программно совместимых, когда машины с точки зрения их технического устройства могут быть весьма непохожими друг на друга, а программы на них пишутся одинаково. Таковыми были, например, машины М-20 (1-е поколение) и БЭСМ-4 (2-е поколение), таковыми являются машины серии ЕС ЭВМ.

Переписать же программу с одного машинного языка на другой чрезвычайно сложно главным образом потому, что, как говорилось выше, трудно разобраться и понять такую программу, и потому, что если она хорошо написана, то весьма тонко учитывает специфику этой первой машины. Гораздо легче было бы создать новую машинную программу, ориентируясь на какую-либо более наглядную запись алгоритма, например блок-схему. Сама же по себе блок-схема позволяет осуществить лишь промежуточный этап при составлении программ, если считать, что конечная цель — программа на языке машины. Да и автоматический перевод алгоритма с описания блок-схем на машинный язык весьма затруднен и, вообще говоря, невозможен. Дело в том, что обычно в блок-схемах опущены многие детали алгоритма, притом такие, которые необходимы для перевода на машинный язык. Правда, можно себе представить очень детальную блок-схему, каждый блок которой соответствует одной машинной команде. Но тогда теряется основное преимущество блок-схем — компактность и наглядность, неперегруженность деталями.

Запись алгоритма на языке блок-схем удобна, как правило, на первом этапе составления программы, когда нужно представить себе алгоритм в целом, его структуру, взаимосвязь его отдельных частей. Этот этап исключительно важен, ибо если допустить ошибку на этой начальной стадии разработки алгоритма, то вся последующая работа над ним будет напрасной. Действительно, зачем нужна детализация

алгоритма и доведение его до уровня машинных команд, если в самом алгоритме содержится ошибка?

Любой алгоритм, как мы знаем, есть последовательность предписаний, выполнив которые, можно перейти от исходных данных за конечное число шагов к результату.

В зависимости от степени детализации предписаний обычно определяется уровень алгоритмического языка — чем меньше детализация, тем более высокого уровня язык. В этом смысле язык блок-схем можно отнести к языкам самого высокого уровня. Но, как мы уже говорили, машине трудно понять такой язык, точнее — трудно преобразовать алгоритм, записанный на языке блок-схем, в программу на машинном языке. Машинные языки и машинно-ориентированные языки в символьных обозначениях — это языки самого низкого уровня, требующие указания мелких деталей процесса обработки данных.

В мире используется реально несколько сотен машинно-независимых алгоритмических языков, предназначенных для описания алгоритмов обработки данных из различных областей науки и техники. Около десятка из них завоевали позиции универсальных языков программирования, нашедших самое широкое распространение и области применения. Многие из таких общепринятых языков стандартизированы и входят как неотъемлемая часть в математическое обеспечение серийно выпускаемых вычислительных машин. К таким языкам, например, относятся: фортран, алгол, кобол, PL-1, лисп, паскаль. Хотя каждый из перечисленных языков имеет свои особенности и свои области применения, где его наиболее целесообразно использовать, у всех у них имеется ряд общих черт или требований, которым они удовлетворяют.

Прежде всего, все предложения таких языков строятся по строгим правилам, обеспечивающим однозначное их понимание. Такой формальный способ построения конструкций языка предопределяет однозначное понимание любого алгоритма, записанного на этом языке, кем бы то ни было, кто знает алгоритмический язык. А так как запись алгоритма понимается однозначно, то можно поручить расшифровку алгоритма и электронной вычислительной машине. Она сама по специальной программе, называемой *транслятором*, переведет запись алгоритма на свой машинный язык.

Алгоритмические языки используют некоторые слова разговорного языка и общепринятые математические символы. Это упрощает понимание смысла действий и описаний в языковых конструкциях, операторах и выражениях.

Другой особенностью машинно-независимых языков, причем весьма важной, является возможность собирать программы из более мелких частей. Такая возможность позволяет упростить разработку больших программ.

Языки программирования — это искусственные языки, созданные



человеком для вполне определенных целей — целей описания алгоритмов обработки данных. Разработчикам языка необходимо четко представлять, для решения какого круга задач будет использоваться алгоритмический язык. Чем более широк этот круг, тем универсальнее должен быть язык. Естественно поэтому, что при создании языка стремятся к тому, чтобы с его помощью иметь возможность представлять как можно более широкий класс алгоритмов. С одной стороны, это хорошо — тогда на этом языке можно записать алгоритм решения практически любой задачи — будь то задача вычислительного характера, задача по обработке каких-либо текстов и т. д. С другой стороны, эта универсальность не позволяет использовать специфику конкретных задач. Запись алгоритма может получиться громоздкой и неудобной для реализации на ЭВМ.

Поэтому стремление к чрезмерной универсальности при построении алгоритмического языка вряд ли оправдано. Все наиболее популярные языки программирования представляют собой некоторый компромисс между универсальностью и ориентацией на удобное задание определенного класса алгоритмов обработки данных. Для каждого круга задач разрабатываются соответствующие языки. Так, например, для решения задач вычислительного характера удобны языки алгол, фортран, бейсик, для экономических задач — кобол.

Если алгоритм записывается на удобном для него языке, то запись получается компактной, наглядной, легко переводится в машинный эквивалент. Следует иметь в виду, что при написании программы на алгоритмическом языке мы сознательно идем на определенные издержки. Во-первых, электронной вычислительной машине потребуется некоторое время для перевода программы на язык машины. Во-вторых, программа после перевода всегда окажется в каком-то смысле хуже, чем программа, написанная опытным программистом сразу же на машинном языке. Ведь человек может лучше учесть специфику системы команд, организацию памяти и другие особенности конкретной задачи, чем транслятор (т. е. программа-переводчик) с универсального алгоритмического языка, не ориентированного на данную ЭВМ и данную задачу.

Однако мы охотно идем на эти издержки, ибо они с лихвой окупаются удобствами для программиста при написании программ.

Программа на алгоритмическом языке пишется несравненно легче и быстрее, получается более компактной, наглядной, чем на языке машины. В ней легко прослеживаются логические связи. Такая программа, как правило, содержит значительно меньше ошибок, да и находить эти ошибки много проще, чем в машинной программе.

Однако и при записи какого-либо алгоритма на алгоритмическом языке нас подстерегает ряд опасностей. Во-первых, можно ошибиться в самой структуре алгоритма, в логической взаимосвязи его частей, в применении алгоритма к тем данным, к которым он неприменим

(например, извлекается квадратный корень из отрицательного числа, происходит деление на ноль, ищется общая мера для стороны квадрата и его диагонали и т. д.). Такие ошибки отыскиваются специальными методами в ходе так называемой отладки программы.

Но могут быть и ошибки, вызванные отступлением от правил написания алгоритмов на алгоритмическом языке. Ведь даже школьник знает, как трудно без ошибок написать, например, сочинение на родном языке, который он знает с рождения. А правила записи алгоритмов на алгоритмическом языке требуют их неукоснительного соблюдения. В противном случае был бы нарушен основной принцип — четкость и строгая однозначность в понимании любой алгоритмической записи. Такие ошибки называют синтаксическими, в отличие от ошибок первого типа, называемых семантическими (смысловыми). Синтаксические ошибки выявляются самой машиной, точнее говоря, транслятором, во время перевода записи алгоритма на язык машины, и машина выдает информацию о том, где произошла ошибка и каков ее характер.

Опыт показывает, что программ без ошибок практически не бывает, разумеется, если эта программа не слишком уж примитивна. Поэтому всегда приходится исправлять как синтаксические, так и семантические ошибки. Но ведь такие же ошибки обязательно есть и в программе, написанной непосредственно на языке машины. Только их больше по количеству, и отыскать их несравненно труднее. Не говоря уж о том, что и писать такую программу тоже очень трудно.

После того как программа, реализующая алгоритм решения задачи, записана на алгоритмическом языке, она должна быть введена в ЭВМ. Для ввода в ЭВМ, как мы уже знаем, служат различные устройства. Программа вместе с исходными данными может быть набита на перфокартах, перфоленте или введена с помощью клавиатуры.

Но ведь машина воспринимает только цифры. Может возникнуть естественный вопрос, а как же ввести в ее память другие символы алгоритмического языка, например круглые скобки или какие-либо слова из набора английских слов, используемых в языке. Делается это следующим образом: каждому такому символу ставится в соответствие вполне определенный набор цифр, как правило, это восьми-разрядный набор двоичных цифр (один байт). Когда вы нажимаете, например, на клавишу клавиатуры, обозначенную буквой А, электронные схемы преобразуют электрический сигнал, идущий от этой клавиши в набор восьми электрических сигналов, обозначающих последовательность нулей и единиц, соответствующих коду этой буквы. Эта последовательность запоминается в определенном месте памяти ЭВМ, таким образом происходит накопление кодов символов в памяти ЭВМ. А в дальнейшем весь таким образом введенный текст

будет переведен транслятором на язык машины. Для этого, конечно, необходимо с помощью специальной команды вызвать транслятор с того языка, на котором написана данная программа, и он переведет ее в рабочую программу, состоящую из элементарных команд машины.

Кроме перевода, как было отмечено выше, транслятор выявляет синтаксические ошибки в программе. После того как все ошибки устранены (это можно сделать, специальным образом редактируя программу), вводятся исходные данные, для которых необходимо получить результат, и программа производит необходимые вычисления.

Отметим, что перевод программы с алгоритмического языка на внутренний язык машина может осуществлять по-разному. Иногда весь текст программы вводится в ЭВМ, переводится транслятором на язык машины, а затем выполняется. Иногда перевод происходит пооперационно, т. е. проверяется правильность записи очередного оператора, и если все правильно, то он переводится в последовательность команд, которая тут же и выполняется. Затем точно так же рассматривается следующий оператор и т.д. Первый способ называется *компиляцией*, второй — *интерпретацией*.

Одним из первых по времени создания алгоритмических языков является язык программирования фортран (FORTRAN=FORMula TRANslation—перевод формул).

В настоящее время это один из самых распространенных алгоритмических языков в мире. Фортран широко применяется при решении различных задач, сопряженных с большим количеством вычислений.

Большой популярностью в мире и у нас в стране пользовался язык алгол (ALGOL=ALGOrithmic Language—алгоритмический язык). Он удобен как для описания вычислительных процессов, так и в качестве средства публикации алгоритмов.

Не менее широкое распространение получили языки PL-1, паскаль, бейсик и ряд других универсальных языков программирования. Трансляторы с этих языков имеются практически на всех работающих в настоящее время ЭВМ.

Появление удобных языков программирования в значительной степени содействовало повсеместному применению ЭВМ в самых различных областях науки, техники и управления производством.

Расширение сферы применения ЭВМ приводит, с одной стороны, к необходимости создавать новые универсальные языки и совершенствовать старые; с другой стороны, потребность в исследовании и решении специфических задач вынуждает создавать специализированные языки программирования. Например, для решения задач обработки экономической информации был создан язык кобол. В нем учитывались особенности обработки коммерческих задач, в которых приходится иметь дело с большим количеством исходных данных и результатов. Причем обычно потребности вычислительного характера

в таких задачах значительно уступают потребностям по всевозможным перемещениям этих данных, их сортировке и поиску.

Все упомянутые выше языки программирования являются машинно-независимыми языками высокого уровня. Изучив какой-либо язык высокого уровня, можно составлять программы для решения задач на любой машине, имеющей транслятор с этого языка.

Описание любого языка, в том числе и алгоритмического, включает в себя алфавит, синтаксис и семантику. Эти понятия уже упоминались выше. Теперь мы их уточним.

*Алфавит* языка — это набор символов, которые могут быть использованы при составлении программы. Клавиатура устройств ввода информации содержит все символы алфавитов, используемых в алгоритмических языках. В частности, как вы знаете, клавиатура, связанная с дисплеем, содержит буквы латинского и русского алфавита, цифры и специальные знаки.

*Синтаксис* определяет правила построения из символов алфавита специальных конструкций, с помощью которых можно составлять различные алгоритмы решения задач.

Систему правил истолкования этих конструкций называют *семантикой* алгоритмического языка. Относительно ЭВМ эти правила истолкования означают, каким образом ЭВМ исполняет шаг алгоритма, описанный соответствующей конструкцией. Например, если фрагмент конструкции имеет вид  $\dots a \times (b + c) \dots$ , то правила семантики предписывают, что сначала производится сложение величин  $b + c$ , а затем умножение результата на  $a$ . Правила семантики (смысла) конструкций обычно вполне естественны, но в некоторых случаях их надо специально оговаривать.

Таким образом, программы, позволяющие однозначно производить процесс переработки данных, составляются с помощью соединения символов из алфавита в соответствии с синтаксическими правилами, определяющими язык, и с учетом правил семантики.

В основе любого алгоритмического языка лежит понятие *оператора*, который представляет самостоятельную единицу языка, описывающую содержание соответствующего этапа алгоритмического процесса, т. е. то самое предписание, последовательность которых задает алгоритм. В языке выделяется фиксированное количество различных типов операторов, каждый из которых определяет некоторый набор действий.

При составлении программ разрешается использовать только эти операторы. По существу, запись любого алгоритма на алгоритмическом языке представляет некоторую последовательность операторов. Они выполняются последовательно, один за другим в порядке их написания. Если же нужно по каким-либо причинам нарушить этот естественный порядок, то для этого используют специальные операторы.

Многие из операторов могут выполнять существенно более крупные действия, чем одна машинная операция какой-либо ЭВМ. Объектом воздействия операторов могут быть числа, переменные, какие-либо тексты, коды, отображающие графическую информацию. Операторы состоят из более мелких конструкций: чисел, выражений, текстов.

Одним из наиболее широко распространенных алгоритмических языков является язык бейсик (BASIC). Его название произошло от сокращения английских слов *Beginner's All-purpose Symbolic Instruction Code*, что в переводе означает «многоцелевой язык символических конструкций для начинающих».

Язык разработан в 1965 г. Особенно часто он используется в *диалоговом* режиме, т. е. режиме, при котором человек, работающий с ЭВМ, сразу же получив ответы на свои запросы, посланные в ЭВМ, в случае необходимости тут же дает ей новые. Происходит как бы диалог «Человек — ЭВМ».

В настоящее время бейсик является одним из ведущих диалоговых языков. В нашей стране он используется на большинстве серийно выпускаемых ЭВМ. Существует много версий языка бейсик. Здесь мы изложим одну из простейших. При работе на любой ЭВМ всегда необходимо учитывать возможные отклонения конкретной версии языка от излагаемого нами варианта.

Основное достижение языка бейсик — простота и в то же время достаточно большие возможности для реализации вычислительных алгоритмов.

Описание языка бейсик и правил составления программ мы начнем с некоторых общих замечаний, которые, вообще говоря, будут относиться к описанию любых других языков. Что значит описать язык? Прежде всего необходимо описать те правила, по которым строятся правильные фразы языка, или, что то же самое, описать правила построения конструкций языка. В свою очередь, что означают термины «правильная фраза», «правильная конструкция»? Они означают то и только то, что правильно построенная конструкция доступна для однозначного понимания. В случае языков программирования речь идет о том, чтобы правильные фразы и правильные конструкции были доступны для однозначного «понимания» компьютером, с одной стороны, и человеком, с другой стороны.

Как же можно описывать правила построения языковых конструкций? В любом учебнике иностранного языка сначала обязательно дается его алфавит, из символов которого строятся все слова, все языковые конструкции. Очевидно, чтобы правильно писать предложения и фразы, надо использовать только знаки заданного алфавита.

При описании конструкций языка бейсик мы будем придерживаться следующего принципа. Малыми латинскими буквами, которые не входят в алфавит языка бейсик, мы будем обозначать основные

языковые понятия, поясняя каждый раз, что эти буквы обозначают. Например, так:

$k \_ n \_ q$

где  $k$  — номер строки,  $n$  — название оператора,  $q$  — содержание оператора. Это нам упростит выписывание схем конструкций и позволит отличать описание конструкции от примеров конкретных конструкций.

## § 5.2. Основные символы языка бейсик. Выражения

Набор основных символов, или алфавит языка бейсик, включает в себя следующие символы:

1) Заглавные буквы латинского алфавита: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z (иногда используют также заглавные буквы русского алфавита).

2) Арабские цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

3) Разделители: , (запятая), ; (точка с запятой), . (точка), ! (двоеточие), ' (апостроф), " (кавычки), ( (открывающая скобка), ) (закрывающая скобка),  $\_$  (пробел).

4) Знаки арифметических операций: + (сложение), — (вычитание), \* (умножение), / (деление), ^ (возведение в степень, который иногда обозначается  $\wedge$  или же  $\wedge$ ).

5) Знаки операций отношений: =,  $\neq$ , >,  $\geq$ , (иногда обозначается  $> =$ ), <,  $\leq$  (иногда обозначается  $< =$ ).

Десятичные числа в бейсике могут записываться как с фиксированной запятой, так и с плавающей. О представлении чисел с фиксированной и плавающей запятой было подробно рассказано в гл. 3. Запись весьма близка к обычной, естественной записи чисел, только вместо привычной нам запятой, разделяющей целую и дробную части, используется точка. Поэтому правильно говорить: «Числа с фиксированной или плавающей точкой».

Числа с фиксированной точкой имеют такую форму записи:

$n$  — для целых чисел (например, 15, 200),

$n.m$  — для чисел, имеющих целую и дробную части (например, —8.1, 0.0061).

Здесь  $n$  и  $m$  — последовательность десятичных цифр, перед которой может стоять знак. Знак «+» у положительных чисел можно опустить.

Возможны также записи вида  $n.$  и  $.m$ , например 10., .52.

Числа с плавающей точкой представляются в форме

$n.m E \pm K$

а также возможны записи

$n. E \pm K$

$.m E \pm K$

Здесь запись *n.m* имеет прежний смысл, запись  $E \pm K$  имеет смысл десятичного порядка числа, а  $K$  может принимать только целые значения.

Примеры записи чисел с плавающей точкой:

$$\begin{aligned} &3.7E+2 \text{ (т. е. } 3.7 \cdot 10^2 = 370), \\ &-12.82E-3 \text{ } (-12.82 \cdot 10^{-3} = -0.01282), \\ &5.E+3 \text{ } (5 \cdot 10^3 = 5000), \text{ } .3E+2 \text{ } (0.3 \cdot 10^2 = 30). \end{aligned}$$

В программах на бейсике употребляются не только константы, но и переменные, значения которых изменяются в ходе выполнения программ. Значениями переменных в бейсике могут быть, вообще говоря, объекты любой природы — числа, тексты, геометрические фигуры и т. д. Мы пока ограничимся простейшей версией языка и будем считать, что в качестве констант и значений переменных могут быть использованы только числа.

Для обозначения переменных используются имена, или *идентификаторы*. В бейсике идентификаторы переменных записываются одной латинской буквой либо латинской буквой, за которой следует цифра. Например,

B, B2, N0

Если бы мы в качестве идентификатора переменной написали NO, то это было бы синтаксической ошибкой, так как идентификатор простой переменной не может состоять из двух букв.

Из определения идентификатора переменной следует, что всего в программе можно использовать не более чем  $26 + 26 \cdot 10 = 286$  различных имен переменных.

Примеры неверной записи идентификатора:

AB (второй символ — буква),

1A (первый символ — цифра),

Ц1 (первая буква — русская),

A10 (слишком длинное имя).

В программах на языке бейсик разрешено использовать массивы. Простейший вид массива — одномерный, когда он представляет собой упорядоченную, пронумерованную последовательность элементов. Это аналог математического понятия — вектор. Например, вектор  $a_1, a_2, a_3, \dots, a_{100}$  есть упорядоченная последовательность из ста элементов. Для того чтобы указать любой из них, достаточно написать имя вектора и индекс, т. е. номер этого элемента.

Именем массива в бейсике может служить только одна латинская буква, которая и является именем или идентификатором массива. Для того чтобы указать какой-либо элемент одномерного массива, необходимо написать идентификатор этого массива, а за ним в круглых скобках индекс, т. е. номер данного элемента в массиве.

Пусть, например, есть массив A и в нем десять элементов. Тогда для указания пятого элемента массива A достаточно написать A(5).

Особо обращаем внимание на непривычную форму записи индексов в алгоритмическом языке. Если в общепринятой математической символике для указания пятого элемента вектора  $A$  мы пишем  $a_5$ , т. е. употребляем подстрочный индекс, то в бейсике пишем  $A(5)$ . Дело в том, что любые символы в памяти ЭВМ располагаются последовательно. Поэтому необходимо, чтобы любой вводимый текст был линейной цепочкой символов. И никаких надстрочных и подстрочных индексов, как и вообще каких-либо надстрочных и подстрочных знаков, в бейсике нет.

В математике часто используется понятие матрицы. Матрица — это таблица, состоящая из строк и столбцов. Матрица обозначается каким-либо именем, чаще всего заглавной буквой. Для того чтобы указать некоторый элемент матрицы, необходимо написать имя этой матрицы (в математической записи обычно пишут вместо заглавной буквы строчную) и два индекса, один из которых является номером строки, а другой — номером столбца, на пересечении которых находится данный элемент.

Выпишем, например, матрицу  $A$ , содержащую три строки и четыре столбца, в обычной математической символике:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}.$$

Если мы хотим указать на элемент матрицы, находящийся во 2-й строке и 3-м столбце, то должны написать  $a_{23}$ .

По существу, матрицу можно считать двумерным массивом — одно измерение по горизонтали, другое — по вертикали. В частном случае, если матрица состоит только из одной строки или только из одного столбца, получается вектор, или одномерный массив.

Как записываются в бейсике элементы одномерных массивов, мы уже показали выше. Теперь покажем, как обозначаются элементы двумерных массивов.

Для указания элемента двумерного массива пишут идентификатор этого массива, а за ним в круглых скобках номер строки и через запятую номер столбца; в которых содержится данный элемент массива.

Например, элемент массива с именем  $M$ , находящийся на пересечении 4-й строки и 2-го столбца этого массива, запишется так:  $M(4, 2)$ . (Заметим, что в обычной математической записи это бы выглядело так:  $m_{42}$ .)

При решении какой-либо задачи, связанной с обработкой массива, каждый элемент этого массива может в ходе выполнения алгоритма принимать различные значения. Поэтому он является переменной величиной. Иногда ее называют переменной с индексами.

Часто в программах удобно задавать не конкретные номера строки



и столбца, как в вышеуказанном примере, а идентификаторы этих номеров или, как еще говорят, идентификаторы индексов. Например,  $M(I, J)$ . Это позволяет, присваивая переменным  $I$  и  $J$  соответствующие значения, выбирать из массива нужные элементы.

Элементы массивов по каждому измерению (по строке или столбцу) нумеруются подряд, причем номер первого элемента всегда равен единице.

Заметим, что прежде чем использовать в программе массивы, их следует специальным образом описать. Как это делается, будет показано позднее.

Как мы уже говорили, основной единицей языка является оператор, задающий то или иное действие описываемого шага алгоритмического процесса. В свою очередь операторы строятся из более мелких конструкций.

Важной конструкцией языка является *выражение*, которое представляет собой формулу, определяющую некоторое значение. На языке бейсик выражение записывается в форме, близкой к обычной математической записи. Оно состоит из чисел, переменных, функций, объединенных знаками арифметических операций.

Выполняются операции в соответствии со следующим приоритетом: сначала вычисляют функции, затем идет возведение в степень, потом умножение и деление и, наконец, сложение и вычитание. Все формулы записываются в строку без надстрочных и подстрочных знаков. Не разрешается ставить подряд два знака операций, например,  $A/-B$ . Нельзя также опускать знаки операций. Ошибочно, например, запись  $2A+B$ . Правильная запись:

$$2 * A + B$$

Если нужно выполнять какие-либо действия с нарушением их приоритета, то необходимо использовать круглые скобки. Например,  $A/(-B+C)$

В этом выражении сначала вычисляется значение знаменателя, а затем произойдет вычисление значения дроби.

К моменту вычисления значения выражения все входящие в него переменные должны иметь какие-либо числовые значения.

Примеры записи выражений

Математическая запись	Запись на бейсике
$\frac{ax^2 + bx + 3.1}{cx^2 + x} \cdot \frac{x^3 - 1}{x^2 - x + 1}$	$(A * X \uparrow 2 + B * X + 3.1) / (C * X \uparrow 2 + X) \\ (X \uparrow 3 - 1) / (X * X - X + 1)$

Если выражение содержит подряд стоящие операции одинакового приоритета, то они выполняются по порядку слева направо. Например, выражение на бейсике

$$A+B/C*D$$

эквивалентно математическому выражению

$$a + \frac{b}{c} \cdot d$$

Если бы мы захотели получить на бейсике выражение, эквивалентное математическому выражению  $a + \frac{b}{c \cdot d}$ , то пришлось бы записать

$$A+B/(C*D)$$

Обратим внимание на типичные ошибки в записи выражений.

1) Нельзя писать два знака операций подряд. Неправильная запись:  $A*-B$ . Правильная запись:  $A*(-B)$ .

2) Нельзя опускать знак умножения. Неправильная запись:  $3X$ . Правильная запись:  $3*X$ .

### § 5.3. Операторы языка бейсик

Программа на языке бейсик записывается в виде последовательности операторов. Каждый из них снабжен числовой меткой, или номером. Номер — это натуральное десятичное число. Запись любого оператора имеет вид

**<метка> <наименование оператора> <содержание оператора>**

(скобки < > используются для выделения понятия языка).

Номера операторов в бейсике играют весьма важную роль, регулируя порядок выполнения операторов. Допускается ситуация, когда операторы в программе записываются с нарушением порядка возрастания их номеров. Тем не менее операторы будут выполняться в строгом соответствии с порядком возрастания их номеров. Например, если написать программу

```
10 <оператор 1>  
40 <оператор 2>  
20 <оператор 3>
```

то в действительности машина выполнит операторы в следующей последовательности:

```
10 <оператор 1>  
20 <оператор 3>  
40 <оператор 2>
```

В тексте программы оператор может занимать только одну строку. И в каждой строке мы будем располагать по одному оператору. Последнее требование мы приводим лишь для простоты написания и чтения программ.

Все операторы в программе обычно нумеруют не подряд, а через десятки. Делается это для того, чтобы впоследствии при каком-либо изменении программы можно было вставить дополнительные операторы в любом месте программы.

Приступим теперь к рассмотрению операторов языка бейсик.

**Оператор присваивания.** Оператор присваивания имеет вид

$$k \sqcup \text{LET} \sqcup v_1 = v_2 = v_3 = \dots = v_n = e$$

Здесь  $k$  — метка, или номер оператора;  $\sqcup$  — пробел (при записи на бумаге какого-либо конкретного оператора знак пробела писать не нужно, нужно лишь оставить вместо него пустое место); LET (пусть) — наименование оператора, говорящее о том, что это именно оператор присваивания;  $v_1, v_2, v_3, \dots, v_n$  — имена переменных,  $e$  — арифметическое выражение.

Оператор присваивания производит следующие действия. Во-первых, вычисляется значение выражения  $e$ , а во-вторых, это вычисленное значение присваивается переменным  $v_1, v_2, v_3, \dots, v_n$ .

Особо обращается внимание на то, что здесь в роли знака присваивания выступает обычный знак равенства, а не знак  $:=$ , который мы использовали до сих пор для обозначения операции присваивания. Мы считаем, что для объяснения сути процесса присваивания какой-либо переменной некоторого значения удобнее использовать именно знак  $:=$ .

В различных алгоритмических языках используются для обозначения присваивания как знак  $:=$ , так и знак  $=$ . Так, в алголе применяется знак  $:=$ , а в фортране, бейсике, PL-1 — знак  $=$ .

Заметим, что в записи алгоритмов на языке блок-схем и в наших пояснениях мы по-прежнему будем использовать для обозначения операции присваивания знак  $:=$ .

Из текста программы на бейсике всегда будет ясно, когда знак  $=$  играет роль операции отношения (т. е. обычный математический смысл этого знака), а когда служит для обозначения операции присваивания.

Примеры операторов присваивания:

10 LET A=5

30 LET B2=C1

20 LET X=Y↑2+2.5/(Y\*Z)

70 LET X=Y=Z=10

(номера операторов выбраны произвольно). В последнем примере всем переменным X, Y и Z присваивается одно и то же значение 10, т. е. этот оператор эквивалентен последовательности операторов

110 LET X=10

120 LET Y=10

140 LET Z=10

Естественно, в бейсике можно применять и указанное выше действие, когда новое значение переменной вычисляется через ее старое

значение. Только теперь это будет, например, такой иметь вид

200 LET  $X = X + 1$

(то, что мы ранее описывали как  $x := x + 1$ ).

С помощью оператора присваивания можно как задавать переменным их начальные значения, так и менять в ходе вычислений эти значения.

**Оператор DATA.** Другой способ задания значений переменных основан на применении блока данных. Блок данных—это упорядоченный числовой массив, который формируется перед началом выполнения программы. Это формирование происходит с помощью операторов DATA (данные), каждый из которых имеет вид

$k \text{ DATA } c_1, c_2, \dots, c_n$

Здесь  $k$ —метка оператора; DATA—наименование оператора;  $c_1, c_2, \dots, c_n$ —числа.

Пусть, например, нужно включить в блок данных числа

5;  $-0.35$ ; 6.2;  $0.3 \cdot 10^{-4}$ .

Для этого запишем оператор

20 DATA 5,  $-0.35$ , 6.2,  $0.3E-4$

Таких операторов, как уже было сказано, в программе может быть несколько, и располагаться они могут в любом месте программы, как, впрочем, и операторы присваивания.

Но блок данных формируется один и данные в него вносятся в порядке очередности операторов DATA. Сначала в блок данных заносятся числа из оператора DATA с наименьшим номером, затем из оператора DATA со следующим по порядку номером и т. д. Завершается блок последним числом из оператора DATA с наибольшим номером. Так, если в программе встречаются операторы

10 DATA 3,  $-5$ , 0.2

.....  
15 DATA 2.7,  $0.6E+3$

.....  
120 DATA 3, 15

в блок данных занесутся 7 чисел: 3;  $-5$ ; 0.2; 2.7; 600; 3; 15. Из примера видно, что одно и то же число может встречаться в блоке данных по несколько раз. Сам блок представляет собой как бы склад, в котором в определенном порядке уложены данные.

**Оператор READ.** Для того чтобы выбрать числа, заложенные в блоке данных, и присвоить их значения некоторым переменным, используется оператор READ (читать). Его общий вид:

$k \text{ READ } v_1, v_2, \dots, v_n$

Здесь  $k$ —метка оператора; READ—наименование оператора;  $v_1, v_2, \dots, v_n$ —имена переменных.

Пусть в программе есть операторы  
 100 DATA 0.2, —5, 16, 0.0005, 114  
 120 READ A, B, C

Тогда с помощью оператора с номером 100 в блок данных занесутся числа 0.2; —5; 16; 0.0005; 114, а в результате выполнения оператора 120 переменной А присвоится значение 0.2, переменной В — значение —5, а переменной С — значение 16. После этого в блоке данных «останутся» числа 0.0005 и 114. Слово «останутся» мы заключили в кавычки, потому что в действительности в блоке данных останутся все числа, но только при выполнении последующих операторов можно присвоить каким-либо переменным лишь оставшиеся незадействованными в данном операторе числа, т. е. 0.0005 и 114. Поэтому, если в программе после указанных выше операторов встретятся операторы

250 READ M

300 READ A

то в момент выполнения оператора 250 переменной М присвоится значение 0.0005, в момент выполнения оператора 300 переменной А присвоится значение 114.

Мы видим, что с помощью оператора 120 переменная А сначала получила значение 0.2, а затем уже с помощью оператора 300 той же переменной А присвоено новое значение 114.

Выбирать из блока данных числа можно лишь подряд, и выборка эта начинается при выполнении первого в программе оператора READ, только начиная с первого числа. Есть некий указатель чисел (или счетчик чисел) в блоке данных, который сначала стоит против первого числа и при считывании каждого числа перемещается к следующему числу. Перед выполнением очередного оператора READ указатель показывает, какое число надо считывать из блока данных и присваивать соответствующей переменной.

Таким образом, используя оператор READ, можно задавать переменным их начальное значение. Причем, если в программе начальные данные заданы с помощью большого количества операторов присваивания, то при изменении начальных значений нужно исправлять в программе все соответствующие операторы присваивания, т. е. придется внести в программу довольно существенные изменения. При использовании оператора READ нужно заменить небольшое количество операторов DATA, ибо в каждом из них может быть сразу несколько новых исходных величин.

С другой стороны, с помощью операторов DATA и READ можно лишь присваивать переменным числовые значения из блока данных, а с помощью операторов LET переменным можно присваивать значения арифметических выражений, например,

LET A = (X \* Y - 1) / Z + 2

Тем самым операторы DATA и READ, с одной стороны, и операторы LET, с другой стороны, не всегда взаимозаменяемы — они дополняют друг друга.

Оператор RESTORE. Мы видели, что операторы READ как бы выбирают из блока данных числовые значения. Иногда бывает необходимо восстановить исходные величины в блоке данных. По существу, для этого надо просто указатель числа в блоке установить на первое число. Эту функцию выполняет оператор RESTORE (восстановить). Его внешний вид:

$k$  RESTORE

Здесь  $k$  — метка; RESTORE — наименование оператора.

В результате работы оператора при выполнении первого же после него оператора READ выборка чисел начнется снова с самого первого числа блока данных.

Оператор RESTORE может находиться в любом месте программы.

Пример:

```
.....  
110 DATA 6,0.3, -12.5  
120 READ A, B, C  
130 LET A = A + B * 2  
135 LET B = B + 5  
140 LET C = B * 3 - 1  
  
.....  
200 RESTORE  
210 READ A, B, C
```

Здесь, как мы видим, в ходе работы программы изменилось значение переменных A, B и C. Если же для последующих действий в программе нужны исходные значения этих переменных, то необходимо их восстановить. Оператор с номером 200 их восстанавливает, т. е. дает возможность использовать числа, начиная с первого в блоке данных.

Теперь представим себе ситуацию в программе, когда нам значения переменных B и C надо восстановить, а переменную A оставить без изменения. В этом случае можно всю приведенную выше программу оставить без изменения, за исключением оператора с номером 210, который теперь будет выглядеть так:

210 READ B, B, C

Здесь переменной B сначала присвоится значение 6, а затем той же переменной B присвоится значение 0.3, т. е. в итоге B получит значение 0.3, а первое присваивание было временным и как бы фиктивным. Мы вынуждены его осуществить, так как присваивание всегда начинаем с самого первого числа блока данных, и число 6 какой-то переменной должно было быть присвоено. Но переменная A получила новое значение, которое мы испортить не хотим, поэтому вме-

сто А мы и написали В. Тем самым для переменной А сохранили ее новое значение, а значение переменной В восстановили. Переменная С с помощью оператора 210 также восстановила свое старое значение — 12.5.

**Оператор безусловного перехода.** Как уже говорилось выше, операторы в программе выполняются последовательно один за другим в порядке возрастания номеров их строк. Но в предыдущих разделах мы не раз имели возможность убедиться в том, что большинство задач не укладывается в узкие рамки линейных схем. И на языке блок-схем, и на языке символьного кодирования мы приводили примеры алгоритмов, в которых приходилось нарушать естественную последовательность действий. Бывают ситуации, когда надо нарушать эту последовательность независимо от каких-либо условий, но бывает и так, что нужно проверять, выполняются или нет некоторые условия, и в зависимости от результата проверки дальнейший ход вычислений направляется по тому или иному пути.

Для реализации безусловного перехода в бейсике используется оператор GOTO (перейти к), общий вид которого:

$k \square \text{GOTO} \square l$

Здесь  $k$  — метка текущего оператора; GOTO — наименование оператора;  $l$  — метка оператора, который будет выполняться непосредственно после выполнения оператора GOTO.

Таким образом, оператор GOTO нарушает естественную последовательность выполнения операторов, и независимо от того, какой оператор идет вслед за  $k$ -й строкой, выполняется оператор с меткой  $l$ . Например,

```
10 LET A=5
20 GOTO 40
30 LET A=1
40 LET A=A+1
```

В этой последовательности операторов есть оператор с меткой 30, в котором переменной А присваивается значение 1. Но этот оператор в данной последовательности не будет выполняться, так как оператор 20 заставит обойти оператор 30 и выполнить сразу оператор 40. Поэтому последовательность действий будет такой — сначала с помощью оператора присваивания 10 переменная А получит значение 5, и вслед за оператором 20 сразу же выполнится оператор 40 и переменной А присвоится значение на единицу больше, т. е. 6.

**Оператор условного перехода.** Посмотрим теперь, как в бейсике осуществляется разветвление вычислительного процесса, когда придется проверять выполнение тех или иных условий. Для этого используют оператор условного перехода IF (если).

Его общий вид:

$k \square \text{IF } l_1 \otimes l_2 \text{ THEN } \square l$

Здесь  $k$  — метка оператора IF; IF — наименование оператора;  $l_1$  и  $l_2$  — выражения;  $\otimes$  — условное обозначение символа отношения ( $=$ ,  $\neq$ ,  $>$ ,  $\geq$ ,  $<$ ,  $\leq$ ); THEN (тогда) — служебное слово;  $l$  — метка оператора, который должен выполняться вслед за данным оператором, если соблюдено условие  $l_1 \otimes l_2$ . Запись  $l_1 \otimes l_2$  мы будем называть условием. Условие считается выполненным (или соблюденным), если удовлетворяется соответствующее отношение. Например, пусть к моменту выполнения условного оператора переменные  $x$  и  $y$  имеют соответственно значения 1 и 2. Тогда условие

$$x+2 > y$$

считается соблюденным (так как  $x+2=3$ ,  $y=2$  и  $3 > 2$ ).

При тех же значениях  $x$  и  $y$  условие

$$x+1 > y$$

уже не соблюдается (так как отношение  $2 > 2$  не справедливо).

Работает оператор IF следующим образом. Проверяется, выполнено ли условие  $l_1 \otimes l_2$ . Если оно нарушено, то после оператора с меткой  $k$  выполняется следующий по порядку оператор. Если же условие  $l_1 \otimes l_2$  выполнено, то вслед за оператором с меткой  $k$  будет выполняться оператор с меткой  $l$ . Например,

```
60 LET A=5
70 IF A > 3 THEN 150
80 LET A=A-1
```

Здесь к моменту выполнения оператора IF переменная A имеет значение 5. Поэтому условие  $A > 3$  оказывается выполненным и следующим по порядку выполняется оператор, находящийся в строке с номером 150.

Заменяем в нашей программе первую строку:

```
60 LET A=3
70 IF A > 3 THEN 150
80 LET A=A-1
```

Здесь к моменту выполнения оператора IF значение переменной A равно 3 и условие  $A > 3$  в операторе IF окажется невыполненным. Поэтому вслед за оператором IF будет выполняться оператор из 80-й строки, т. е. просто следующий по порядку.

Теперь в нашем распоряжении уже достаточно средств, чтобы попытаться составить программы, реализующие некоторые простые алгоритмы. Будем считать в последующих примерах, что наши программы являются частью некоторой более общей программы. Поэтому пока у нас остаются в стороне вопросы ввода и вывода информации и некоторые другие вопросы организации программ.

Пусть, например, нам нужно вычислить значение абсолютной величины переменной  $x$  и присвоить вычисленное значение



переменной  $y$ . Как известно,

$$y = |x| = \begin{cases} x, & \text{если } x \geq 0, \\ -x, & \text{если } x < 0. \end{cases}$$

Блок-схема алгоритма представлена на рис. 5.1.

Составим программу на бейсике:

```
40 IF X ≥ 0 THEN 70
50 LET Y = -X
60 GOTO 80
70 LET Y = X
80 ...
```

(Чтобы показать, что наша программа является лишь фрагментом некоторой более общей программы, мы поставили многоточие в опера-

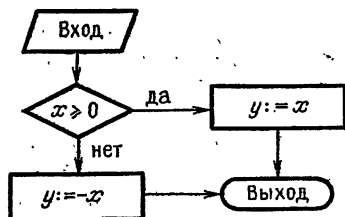


Рис. 5.1

торе с номером 80.) В нашей программе проверяется условие  $X \geq 0$ . Если текущее значение переменной  $X$  окажется меньше нуля, т. е. условие не выполняется, то сохраняется естественный порядок следования операторов и выполняется оператор  $\text{LET } Y = -X$ , т. е. переменной  $Y$  присвоится значение  $-X$ . Вслед за этим оператор

перехода  $\text{GOTO}$  сразу же отправит в конец нашей программы — на 80-ю строку, т. е. к этому моменту значение  $Y$  окажется равным  $-X$ , что и требовалось в данном случае. Если же текущее значение переменной  $X$  будет больше или равно нулю, то условие в операторе  $\text{IF}$  оказывается выполненным и вслед за оператором  $\text{IF}$  будет работать сразу же оператор из 70-й строки (операторы 50 и 60 будут пропущены):

```
70 LET Y = X
```

т. е. переменной  $Y$  присвоится текущее значение переменной  $X$ , а затем будет выполнен следующий по порядку оператор (из 80-й строки).

Ту же задачу можно решить и с помощью более короткой программы:

```
100 LET Y = -X
110 IF X < 0 THEN 120
115 LET Y = X
120 ...
```

Здесь переменной  $Y$  сразу же присваивается значение  $-X$  независимо от знака текущего значения переменной  $X$  и лишь затем проверяется знак этой переменной. Если окажется, что  $X < 0$ , то сразу же будет выполняться оператор из 120-й строки, т. е. к этому моменту переменная  $Y$  окажется равной  $-X$ , что и требовалось. Если

же в действительности оказалось, что текущее значение  $X$  к началу программы неотрицательно, то мы напрасно присвоили переменной  $Y$  значение  $-X$ . Но это поправимо, так как проверка условия  $X < 0$  в операторе IF покажет, что условие не соблюдено, и, следовательно, сохраняется естественный порядок выполнения операторов. Значит, вслед за оператором IF выполнится следующий по порядку оператор:

115 LET Y = X

который исправит значение переменной  $Y$  на то, которое должно быть в данном случае (случай  $X \geq 0$ ), т. е. переменной  $Y$  присвоит значение  $X$ .

Даже на таком простом примере мы убедились в том, что решение одной и той же задачи может быть реализовано различными программами.

Оператор IF может разветвлять алгоритмический процесс только по двум направлениям. Но ведь часто бывает необходимость в организации нескольких ветвей алгоритма. Тогда приходится применять оператор IF несколько раз.

Пусть, например, надо вычислить значение функции, заданной следующим образом: переменная  $Y$  принимает значение 1, если  $X > 0$ , значение 0, если  $X = 1$ , и значение  $-1$ , если  $X < 0$ . В математике такую функцию обозначают  $Y = \text{sign } X$ . Блок-схема алгоритма представлена на рис. 5.2.

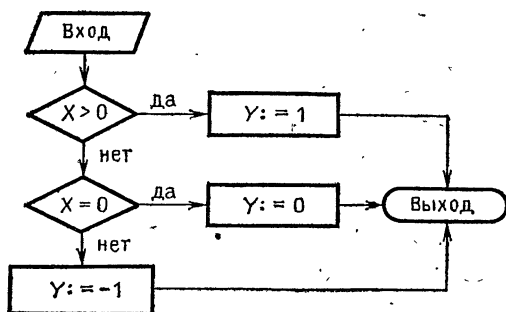


Рис. 5.2

Приведем программу на бейсике:

```

10 IF X > 0 THEN 70
20 IF X = 0 THEN 50
30 LET Y = -1
40 GOTO 80
50 LET Y = 0
60 GOTO 80
70 LET Y = 1
80 ...
  
```

Здесь при выполнении условия  $X > 0$  осуществляется переход на 70-ю строку, в которой результату присваивается значение 1, и затем выход из нашей программы к оператору под номером 80.

Если же условие  $X > 0$  не выполнено, то происходит переход к следующей строке — к 20-й, где переменная  $X$  сравнивается с нулем. Если  $X = 0$ , то вслед за 20-й строкой выполнится оператор из 50-й строки и в результате  $Y$  получит нулевое значение, после чего выполнится переход на конец нашей

программы — на 80-ю строку.

Наконец, если условие  $X = 0$  тоже окажется невыполненным, то после 20-й строки будет выполняться оператор из следующей — 30-й строки, где  $Y$  примет значение  $-1$ , а затем оператор GOTO отправит сразу на выход из этого фрагмента программы.

Пользуясь перечисленным выше набором операторов, мы можем теперь писать программы и для алгоритмов, реализующих циклические процессы.

Пусть нам надо вычислить таблицу значений функции  $y = x^3$  на

отрезке  $[0; 1]$  с шагом по  $x$ , равным  $h = 0.01$ . Начальное значение  $x_{\text{нач}} = 0$ , конечное —  $x_{\text{кон}} = 1$ .

Для записи алгоритма на языке блок-схем и на бейсике введем следующие обозначения:

текущее значение  $x$  — через  $X$ ;

вычисленное значение функции  $y = x^3$  на  $i$ -м шаге циклического процесса — через  $Y(I)$ ;

величину шага  $h$  — через  $H$ ;

$x_{\text{кон}}$  — через  $X1$ .

(Вспомним, что алфавит языка бейсик не допускает использование маленьких букв.)

Блок-схема алгоритма представлена на рис. 5.3.

Программа на бейсике примет вид:

```
10 DATA 0., 1., 0.01, 1
20 READ X, X1, H, 1
30 IF X > X1 THEN 80
40 LET Y(I) = X↑3
```

```

50 LET X=X+N
60 LET I=I+1
70 GOTO 30
80 ...

```

Прокомментируем программу. Первые две строки задают переменным  $X$ ,  $X1$ ,  $H$ ,  $I$  начальные значения. Оператор IF проверяет условие — не вышло ли текущее значение переменной  $X$  за границу интервала, т. е. не стало ли оно больше единицы. И если условие не выполнено, то происходит переход к следующему по порядку оператору, который вычисляет значение  $Y(I)=0\uparrow 3$ . Затем значение переменной  $X$  увеличивается на  $H$ , в данном случае на  $0.01$ ,  $I$  увеличивается на единицу, т. е. после первого выполнения оператора 60  $I$  станет равным двум. И, наконец, оператор GOTO вернет нас к оператору IF. Теперь уже с конечным значением  $X1$  сравнивается новое значение переменной  $X$ . Оно по-прежнему меньше, чем  $X1$ . Поэтому вычисляется значение  $Y(I)=0.01\uparrow 3$ . Затем опять  $X$  увеличивается на  $H$ ,  $I$  — на единицу. Снова оператор GOTO вернет к оператору 30 и т. д.

Наконец,  $X$  станет равным конечному значению, в нашем случае единице. Условие  $X > X1$  еще не будет выполнено (единица не больше, чем единица). Поэтому в последний раз вычислится значение  $Y(I)$ .  $X$  еще раз увеличится на  $H$  и станет больше, чем  $X1$ ,  $I$  в последний раз увеличится на единицу, оператор GOTO отправит нас на 30-ю строку. И здесь уже условие  $X > X1$  окажется выполненным. А потому произойдет переход на 80-ю строку, т. е. на выход из нашей программы.

Число повторений цикла в рассмотренном алгоритме зависит от размеров интервала изменения  $X$  и шага  $H$  — чем больше интервал и меньше значение  $H$ , тем больше количество повторений цикла. Для того чтобы изменить эти параметры, достаточно в операторе DATA задать новые числа, т. е. в программе изменится лишь одна строка.

**З а м е ч а н и е.** В приведенном примере есть одна тонкость. Дело в том, что если все вычисления производятся в двоичной системе счисления, то может случиться, что цикл для вычисления табличных значений сработает на один раз меньше, чем нужно. В самом деле, числа 0 и 1 имеют точное представление в двоичной системе, а число  $0.01$  представить точно конечным числом цифр в двоичной системе невозможно. Чтобы убедиться в этом, переведем  $0.01$  в двоичную систему:

0	0	0	0	0	0	0	0	1	0	1	0	0	0	1	...
01	02	04	08	16	32	64	28	56	12	24	48	96	92	...	

Таким образом, получим  $(0.01)_{10} = (0.0000010100011\dots)_2$ .

В зависимости от того, сколько в ячейке памяти отводится разрядов под мантиссу, число 0.01 будет представлено в двоичной системе счисления округленным—либо с недостатком, либо с избытком. Если с недостатком, то при значении  $X$ , примерно равном единице (получится чуть меньше, чем единица), еще выполнится оператор  $LET Y = X \uparrow 3$ ; если же с избытком, то при последнем прибавлении  $X = X + H$  окажется, что  $X$  чуть больше единицы и условие  $X > X1$

выполнится, т. е. произойдет выход из программы, а значение  $Y$  при  $X = 1$  так и не вычислится.

Чтобы избежать этой неприятности, можно, например, оператор в 30-й строке изменить следующим образом:

30 IF  $X > X1 + H/2$  THEN 80

Так как погрешность округления заведомо меньше, чем  $H/2$ , то при последовательном выполнении оператора  $X = X + H$ , когда  $X$  будет примерно равен 1 (с точностью до погрешности округления), условие  $X > X1 + H/2$  не выполнится и обязательно при этом  $X$  вычислится значение  $Y$ . Последующее выполнение оператора  $X = X + H$

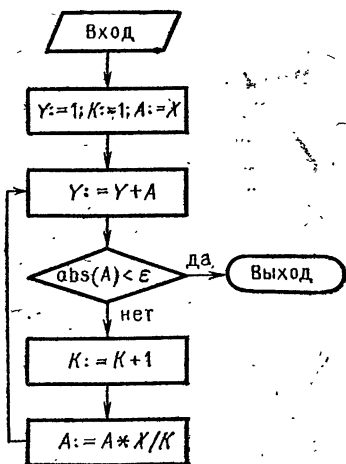


Рис. 5.4

уже выведет  $X$  за границу  $X1 + H/2$ , и произойдет выход из цикла. Еще проще осуществлять проверку на окончание цикла, если сравнивать значение переменной  $I$  с ее конечным значением, так как  $I$  принимает лишь целые значения.

Как известно (гл. 2), многие элементарные функции вычисляют, разлагая их в бесконечные суммы. Например, функцию вида  $y = e^x$  можно представить в виде

$$y = e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots$$

Мы зададим некоторое достаточно малое положительное значение  $\varepsilon$  и будем считать, что заданная точность вычисления функции достигнута, если последнее слагаемое в накапливаемой сумме оказалось по абсолютной величине меньше  $\varepsilon$ . Тогда мы прекращаем суммирование и остальные слагаемые (хотя их и бесконечное количество) попросту отбрасываем.

Блок-схема алгоритма представлена на рис. 5.4.

В предложенном алгоритме результат присваивается переменной  $Y$ , в роли очередного слагаемого выступает переменная  $A$ , переменная  $K$  служит для вычисления знаменателя в очередном  $K$ -м слагаемом— $K!$

Процесс, очевидно, циклический. Число повторений цикла зависит от значения  $X$  и точности вычислений  $\varepsilon$ . Чем больше значение абсолютной величины  $X$  и меньше  $\varepsilon$ , тем большее количество раз повторится цикл, т. е. тем большее количество слагаемых придется брать в сумме для достижения заданной точности результата.

Составим программу, реализующую тот же алгоритм на языке бейсик.

Введем обозначения:  $Y$  — результат;  $X$  — аргумент функции;  $E$  — заданная точность вычислений (т. е. то, что в блок-схеме у нас обозначалось буквой  $\varepsilon$ , но греческих букв нет в алфавите языка бейсик;  $A$  — очередное слагаемое в сумме;  $K$  — целочисленное значение, служащее для вычисления знаменателя в очередном слагаемом). Как обычно, будем считать, что  $X$  и  $E$  заданы не в нашей программе, а в какой-то другой, в которую наша программа включена как часть.

```
110 LET Y=1
120 LET K=1
130 LET A=X
140 LET Y=Y+A
150 IF A ≥ 0 THEN 180
160 LET A1=-A
170 GOTO 190
180 LET A1=A
190 IF A1 < E THEN 230
200 LET K=K+1
210 LET A=A*X/K
220 GOTO 140
230 ...
```

Легко видеть, что программа на бейсике составлена в точном соответствии с блок-схемой. Нам пришлось только использовать вспомогательную переменную  $A1$  для вычисления абсолютной величины переменной  $A$ .

На языке блок-схем и машинно-ориентированном языке мы уже знакомимся с приемами составления программ для алгоритмов решения задач, в которых используются вложенные циклы, т. е. внутри одного цикла содержится другой цикл. Составим программу с вложенными циклами и на языке бейсик.

Пусть нужно получить таблицу значений  $y=e^x$  для значений переменной  $x$ , изменяющейся от 0 до 10 с шагом  $h=0.05$  и точностью  $\varepsilon=0.0001$ . Сначала представим алгоритм в виде блок-схемы (используем прежние обозначения; см. рис. 5.5).

Во внутреннем цикле вычисляется значение функции  $y=e^x$  для одного значения  $x$ , а внешний цикл управляет изменением переменной  $x$ , и каждое новое значение  $e^x$  для очередного значения  $x$  присваивается очередной компоненте массива результатов  $Y(I)$ .

Так как функцию надо в нашем случае вычислить в 201-й точке, то значение  $I$  сравнивается с его конечным значением — 201.

Теперь реализуем алгоритм на бейсике:

```
50 DATA 0, 1, 0.05, 10, 0.0001
60 READ X, I, H, X1, E
70 LET Z=1
80 LET A=X
90 LET K=1
100 LET Z=Z+A
110 IF A ≥ 0 THEN 140
120 LET A1=-A
130 GOTO 150
140 LET A1=A
150 IF A1 < E THEN 190
160 LET K=K+1
170 LET A=A * X/K
180 GOTO 100
190 LET Y(I)=Z
200 LET X=X+H
210 LET I=I+1
220 IF I ≤ 201 THEN 70
230 ...
```

Эта программа, как и предыдущая, составлена в полном соответствии с блок-схемой алгоритма. Стрелками мы отметили внутренних и внешний циклы. В результате работы программы получим набор значений  $y = e^x$ , где каждое очередное значение  $X$  отличается от предыдущего значения на  $H$ . Если нам нужно получить таблицу значений функции  $y = e^x$  на другом интервале  $X$  и с другим шагом изменения переменной  $X$ , т. е. с другим  $H$ , то для этого достаточно изменить соответствующие числа в операторе DATA и константу в условии в строке 220.

**Оператор цикла.** Для организации циклических процессов существует более удобный способ, чем использование оператора IF. Как мы видели из предыдущих примеров, в циклических программах есть переменные, меняющие свои значения в каждом новом шаге повторения. От значения одной из них зависит, надо ли делать очередной шаг циклического процесса или же нужно выйти из цикла. Такую переменную, управляющую количеством повторений цикла, называют параметром цикла. Так, в последнем примере переменная  $I$ , управляющая количеством повторений внешнего цикла при вычислении таблицы значений функции  $y = e^x$ , являлась параметром этого внешнего цикла.

Применяют специальный оператор цикла, который носит название FOR (для). За словом FOR в записи оператора следует специ-

ального. вида информация, в которой указывается параметр цикла и правила его изменения в ходе циклического процесса. Эта информация составляет так называемый заголовок цикла.

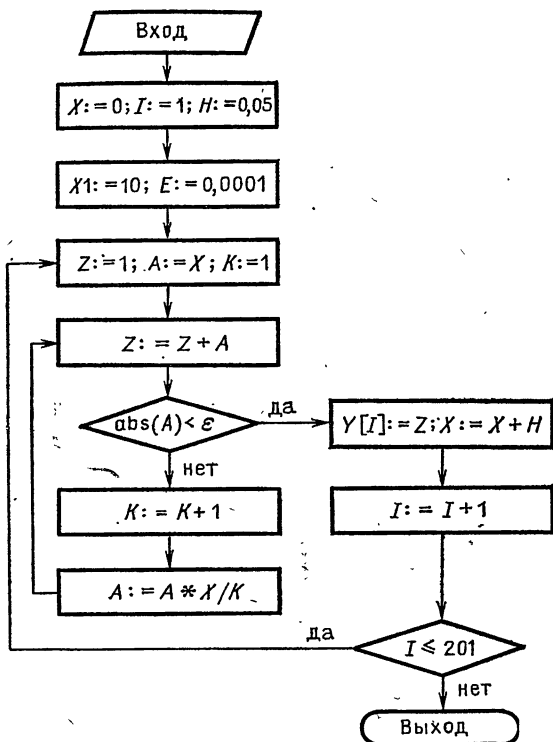


Рис. 5.5

Общий вид заголовка цикла FOR следующий:

$k \text{ FOR } v = e_1 \text{ TO } e_2 \text{ STEP } e_3$

Здесь  $k$  — метка данного оператора; FOR — наименование оператора;  $v$  — имя параметра цикла; TO (до) — служебное слово;  $e_1$  — выражение, определяющее начальное значение параметра цикла  $v$ ;  $e_2$  — выражение, определяющее конечное значение  $v$ ; STEP (шаг) — служебное слово;  $e_3$  — выражение, задающее величину  $\Delta v$  — приращение параметра цикла  $v$ .

В случае, если  $v$  — целое число и  $e_3 = 1$ , то заголовок цикла можно записать более коротко:

$k \text{ FOR } v = e_1 \text{ TO } e_2$

Вслед за заголовком цикла идет последовательность операторов, которая составляет тело цикла. Заканчивается тело цикла оператором



NEXT (следующий). Его общий вид:

$k \sqcup \text{NEXT} \sqcup v$

Здесь  $k$  — метка оператора NEXT; NEXT — наименование оператора;  $v$  — имя параметра цикла (оно обязательно должно совпадать с именем параметра цикла в заголовке оператора FOR).

Итак, цикл организуется следующим образом:

$k \sqcup \text{FOR } v = e_1 \sqcup \text{TO } e_2 \sqcup \text{STEP } e_3$

$\dots S \dots$

$m \sqcup \text{NEXT} \sqcup v$

Здесь символом  $S$  мы условно обозначили тело цикла, т. е. ту последовательность операторов, которая должна многократно выполняться. А количеством повторений цикла и изменением параметра цикла управляет заголовок цикла. Происходит это следующим образом. Вычисляются значения выражений  $e_1$ ,  $e_2$  и  $e_3$ . Параметру цикла  $v$  присваивается значение  $e_1$ . И при этом значении параметра  $v$  выполняются операторы, составляющие тело цикла.

По достижении оператора NEXT  $v$  происходит изменение параметра цикла  $v$  — к нему добавляется приращение  $\Delta v$ , задаваемое значением выражения  $e_3$  ( $v := v + e_3$ ). Это новое значение сравнивается с конечным значением параметра цикла, равным  $e_2$ . Отметим, что значения выражений  $e_1$ ,  $e_2$ ,  $e_3$  могут быть любыми числами, как положительными, так и отрицательными.

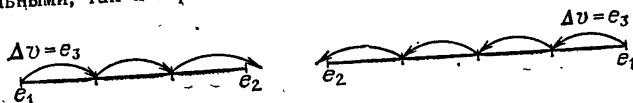


Рис. 5.6

Если  $e_3 > 0$  и  $v \leq e_2$  или же  $e_3 < 0$  и  $v \geq e_2$ , то операторы тела цикла выполняются снова. Затем опять к параметру  $v$  добавляется приращение  $\Delta v$ , равное  $e_3$ , и новое значение  $v$  сравнивается с  $e_2$  и т. д. Цикл повторяется до тех пор, пока при  $e_3 > 0$  параметр цикла  $v$  не станет больше  $e_2$  или же при  $e_3 < 0$  не станет строго меньше  $e_2$  (рис. 5.6).

Независимо от значений выражений  $e_1$ ,  $e_2$  и  $e_3$  операторы тела цикла обязательно выполняются хотя бы один раз. По выходе из цикла параметр цикла сохраняет последнее значение, использованное в цикле.

Выход из цикла может произойти как вследствие того, что параметр цикла исчерпал все свои возможные значения, так и в результате выполнения оператора перехода, если таковой окажется в теле цикла.

Работу оператора цикла FOR можно проиллюстрировать блок-схемой (рис. 5.7).

На бейсике тот же процесс можно реализовать следующей последовательностью операторов:

```
10 LET V=E1
```

```
    . . . . .
{ K   S
```

```
220 LET V=V+E3
```

```
230 IF E3*(V-E2) ≤ 0 THEN K
```

Символом S мы обозначили последовательность операторов, соответствующую телу цикла. Первый из этих операторов помечен меткой K (метки остальных операторов опущены).

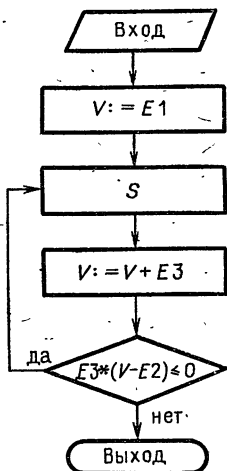


Рис. 5.7

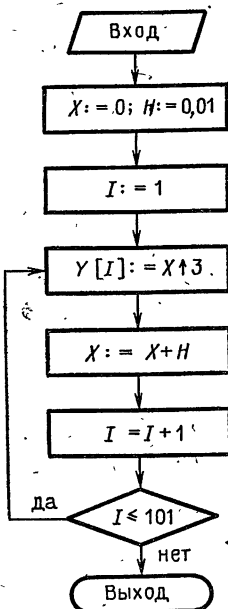


Рис. 5.8

Примеры. 1. Вычислить таблицу значений  $y = x^3$  на отрезке  $[0; 1]$  с шагом 0,01:

```
10 LET X=0
```

```
20 LET H=0.01
```

```
30 FOR I=1 TO 101
```

```
40 LET Y(I)=X+3
```

```
50 LET X=X+H
```

```
60 NEXT I
```

Программа получилась более компактной и наглядной, чем программа решения той же задачи с применением оператора IF (см. с. 186).

Проиллюстрируем ее блок-схемой (рис. 5.8).

2. Вычислить значение полинома  $y = P_n(x) = a_0x^{30} + a_1x^{29} + \dots + a_{30}$ . Его удобнее вычислять по схеме Горнера:

$$y = (\dots((a_0x + a_1)x + a_2)x + \dots + a_{29})x + a_{30}.$$

Пусть вне нашей программы уже заданы значения коэффициентов полинома и  $x$ , для которого надо вычислить полином.

Введем обозначения:  $X$  — аргумент полинома;  $A(1), A(2), \dots, A(31)$  — соответственно коэффициенты полинома  $a_0, a_1, \dots, a_{30}$  (нумерация сдвинута на единицу, так как в бейсике нумерация элементов массива начинается обязательно с единицы).

Программа на бейсике может выглядеть, например, так:

```
50 LET Y=A(1)
60 FOR I=2 TO 31
70 LET Y=Y*X+A(I)
80 NEXT I
```

Здесь в теле цикла присутствует оператор присваивания, который будет многократно выполняться. Сначала, при значении параметра цикла  $I$ , равном двум, переменной  $Y$  присвоится значение  $A(1) * X + A(2)$  (т. е.  $a_0x + a_1$ ). Затем  $I$  увеличится на единицу и станет равным трем. Так как условие  $3 \leq 31$  соблюдается, то оператор присваивания в 70-й строке снова выполнится и переменная  $Y$  получит значение  $(A(1) * X + A(2)) * X + A(3)$  (т. е.  $(a_0x + a_1)x + a_2$ ). И так далее, пока, наконец, при  $I$ , равном 31, оператор присваивания выполнится в последний раз и переменная  $Y$  получит искомое значение.

3. Вычислить приближенное значение функции  $y = e^x$ , просуммировав для этого 50 слагаемых:

$$y = e^x = 1 + x + \frac{x^2}{2!} + \dots + \frac{x^{50}}{50!}.$$

Воспользовавшись теми же обозначениями, что были в примере на с. 189, напомним программу

```
100 LET Y=1
110 LET A=X
120 FOR K=2 TO 51
130 LET Y=Y+A
140 LET A=A*X/K
150 NEXT K
```

Приведем блок-схему алгоритма, иллюстрирующую нашу программу (рис. 5.9).

Сделаем следующие замечания:

1. Вход в тело цикла, минуя заголовок цикла, запрещен, т. е. нельзя попасть ни на какой оператор тела цикла извне с помощью операторов GOTO или IF. Ибо в таком случае были бы не определены значения выражений  $e_1$ ,  $e_2$  и  $e_3$ .

2. Значения  $e_1$ ,  $e_2$  и  $e_3$  вычисляются единожды перед началом выполнения тела цикла. В теле цикла они уже перевычисляться не могут, т. е. начальное и конечное значения параметра цикла, а также его приращение в том же цикле изменяться не могут, даже если там меняются значения входящих в выражения  $e_1$ ,  $e_2$  и  $e_3$  переменных. В то же время сам параметр цикла можно изменить в теле цикла и помимо добавления приращения  $e_3$ .

Пример. Пусть нам надо вычислить сумму  $y = x_1 + x_2 + \dots + x_{20} + x_{22} + \dots + x_{40}$ , т. е. первые двадцать чисел массива суммировать подряд, а затем суммировать только элементы, стоящие на четных местах:

```
30 LET Y=0
40 FOR I=1 TO 40
50 IF I ≤ 20 THEN GOTO 60
60 LET I=I+1
70 LET Y=Y+X(I)
80 NEXT I
```

Здесь цикл повторяется не 40 раз, а 30, так как, начиная с  $I=21$ , кроме обычного добавления с помощью оператора NEXT шага приращения (в нашем случае он равен 1) в том же цикле выполняется оператор 60, который каждый раз дополнительно увеличивает параметр цикла  $I$  на единицу. Тем самым  $I$ , начиная со значения 20, будет увеличиваться каждый раз на 2.

3. Параметр цикла, вообще говоря, может и не фигурировать в теле цикла. Пусть, например, нам нужно вычислить  $y = x^{20}$ . Воспользуемся для этого оператором FOR:

```
60 LET Y=1
70 FOR I=1 TO 20
80 LET Y=Y*X
90 NEXT I
```

Здесь параметр цикла нужен только для управления числом повторений цикла. Он лишь «следит» за тем, чтобы оператор присваивания  $Y = Y * X$  выполнялся ровно 20 раз.

4. Во всех рассмотренных нами примерах с использованием оператора цикла в заголовке оператора цикла не указывается шаг изменения параметра цикла, так как он всегда был равен единице.

Рассмотрим пример, где шаг изменения параметра цикла отличается от единицы. Пусть нам надо среди чисел  $X_1, X_2, X_3, \dots, X_{40}$  просуммировать только неотрицательные числа, стоящие на четных позициях (с четными индексами), начиная от числа с самым большим

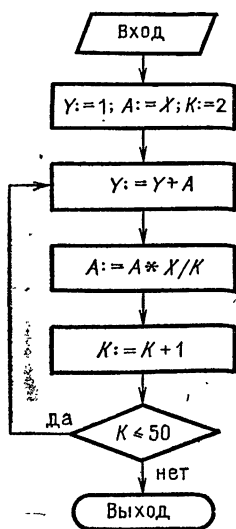


Рис. 5.9

индексом (в нашем случае 40) и до первого встретившегося среди них отрицательного числа. В частности, если уже число  $X_{40}$  отрицательно, то положить искомую сумму равной нулю.

Составим блок-схему алгоритма (рис. 5.10).

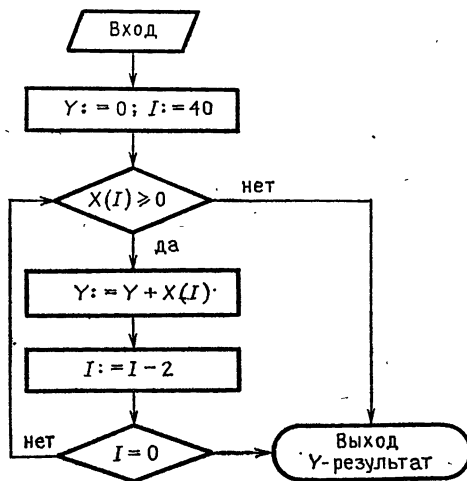


Рис. 5.10

Соответствующая программа на бейсике выглядит следующим образом:

```

40 LET Y=0
50 FOR I=40 TO 2 STEP -2
60 IF X(I) < 0 THEN 90
70 LET Y=Y+X(I)
80 NEXT I
90 ...
  
```

В нашей программе возможны два случая окончания циклического процесса. Если все числа, стоящие на четных позициях, неотрицательны, то цикл закончится, когда параметр цикла  $I$  примет все значения от 40 до 2 (четные). Если же среди рассматриваемых чисел последовательности встретится отрицательное число, то суммирование прекратится, числа в последовательности больше просматриваться не будут и произойдет выход из цикла при том значении параметра цикла  $I$ , при котором встретилось первое по порядку просмотра отрицательное число  $X(I)$ .

С использованием операторов цикла можно организовать и вложенные циклы. Так, для трех вложенных циклов структура про-

граммы выглядит следующим образом:

```

[ k1 ⊐ FOR v1 = ...
  [ k2 ⊐ FOR v2 = ...
    [ k3 ⊐ FOR v3 = ...
      . . . . .
    [ m3 ⊐ NEXT v3
      [ m2 ⊐ NEXT v2
        [ m1 ⊐ NEXT v1

```

Пример. Всем элементам матрицы  $A$ , состоящей из двух строк и трех столбцов, присвоить значение 1:

```

10 FOR I=1 TO 2
20 FOR J=1 TO 3
30 LET A(I,J)=1
40 NEXT J
50 NEXT I

```

Оператор из 10-й строки задает параметру  $I$  начальное значение, равное 1. При этом значении  $I$  выполнится внутренний цикл, т. е. операторы из строк 20, 30 и 40.

В результате их работы всем элементам 1-й строки, т. е.  $A(1, 1)$ ,  $A(1, 2)$  и  $A(1, 3)$ , присвоится значение 1. Параметр  $J$  при этом пробежит все значения 1, 2 и 3. Затем оператор  $NEXT I$  увеличит значение параметра внешнего цикла  $I$  на 1,  $I$  станет равным 2 и снова выполнится внутренний цикл; причем параметр внутреннего цикла  $J$  снова поочередно примет значения 1, 2 и 3, а значит, всем элементам 2-й строки, т. е.  $A(2, 1)$ ,  $A(2, 2)$  и  $A(2, 3)$ , тоже присвоится значение 1.

Оператор  $FOR$  удобен для организации циклических процессов с заранее известным числом повторений цикла. Однако часто приходится иметь дело с вычислительными процессами, где число повторений цикла заранее неизвестно. Так происходит, например, когда необходимо вычислить значение некоторой элементарной функции с заданной точностью, разложив ее в бесконечную сумму.

Вспомним, как мы вычисляли значение функции  $e^x$  с заданной точностью  $\varepsilon > 0$ . Там число слагаемых в разложении функции  $e^x$  в бесконечную сумму, необходимое для достижения заданной точности, зависело, во-первых, от этой точности, а во-вторых, от значения  $x$ . Цикл был организован с помощью операторов  $IF$  и  $GOTO$ . Но такие циклы можно записывать и более удобным способом — с помощью оператора  $WHILE$  (пока).

Заголовок этого оператора имеет вид

$k \sqsubseteq WHILE \sqsubseteq e_1 \otimes e_2$

Здесь  $k$  — метка оператора;  $WHILE$  — название оператора;  $e_1 \otimes e_2$  — условие, оно имеет тот же смысл, что и в операторе  $IF$ ;  $e_1$  и  $e_2$  — выражения;  $\otimes$  — знак отношения, а условие  $e_1 \otimes e_2$  выполнено, если отношение  $e_1 \otimes e_2$  справедливо при текущих значениях входящих в него переменных.

Вслед за заголовком цикла WHILE идет последовательность операторов, составляющая тело цикла. Она-то и должна многократно выполняться. Заканчивается тело цикла оператором

$k \sqcup \text{WEND}$

Здесь  $k$  — метка оператора; WEND — наименование оператора. Таким образом, цикл организуется следующим образом:

$k \sqcup \text{WHILE } e_1 \otimes e_2$   
 $\dots S \dots$   
 $m \sqcup \text{WEND}$

Символом  $S$  мы условно обозначили последовательность операторов, входящих в тело цикла.

Рассмотрим теперь работу оператора WHILE. Проверяется, соблюдается ли условие  $e_1 \otimes e_2$ . Если соблюдается, то выполняется последовательность операторов тела цикла. Затем снова проверяется справедливость отношения  $e_1 \otimes e_2$ . Если оно справедливо, то снова выполняются операторы тела цикла. Снова проверяется условие и т. д. До тех пор, пока условие не перестанет соблюдаться. Как только это произойдет, закончится выполнение цикла.

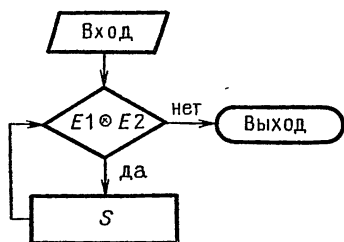


Рис. 5.11

Может возникнуть вопрос: все время проверяется одно и то же условие —  $e_1 \otimes e_2$ , почему же оно то соблюдается, то перестает соблюдаться? Дело в том, что операторы тела цикла обычно меняют значения операндов, входящих в выражения  $e_1$  и  $e_2$ . Поэтому после каждого выполнения операторов тела цикла могут изменяться значения выражений  $e_1$  и  $e_2$ , а следовательно, и справедливость условия  $e_1 \otimes e_2$ . Если при первой же проверке окажется, что условие не выполнено, то цикл не выполнится ни разу.

Работу оператора цикла WHILE можно проиллюстрировать блок-схемой (рис. 5.11).

Составим теперь программу для вычисления функции  $y = e^x$  в тех же обозначениях, что и при решении задачи с помощью оператора IF (с. 189). Предположим для простоты, что  $x \geq 0$ :

```

110 LET Y=1
120 LET K=1
130 LET A=X
140 WHILE A ≥ E
150 LET Y=Y+A
160 LET K=K+1
170 LET A=A*X/K
180 WEND
  
```

Задача вычисления таблицы значений функции  $y=e^x$  при  $x \geq 0$  (с. 190) с помощью операторов FOR и WHILE может быть записана так:

```
50 DATA 0, 1, 0.05, 10, 0.0001
60 READ X0, I, H, X1, E
70 FOR I=1 TO 201
80 LET Y(I)=1
90 LET K=1
100 LET A=X
110 WHILE A ≥ E
120 LET Y(I)=Y(I)+A
130 LET K=K+1
140 LET A=A*X/K
150 WEND
160 LET X=X+H
170 NEXT I
180 ...
```

В программе используются вложенные циклы. Внутренний — WHILE — служит для вычисления функции  $e^x$  для одного значения  $x$ , внешний — FOR — управляет изменением значения аргумента  $x$ , чтобы функция  $e^x$  вычислялась в различных точках отрезка  $[X_0; X_1]$ :

$X_0, X_0+H, X_0+2H, \dots, X_1$

Результаты вычислений составят массив переменных  $Y(1), Y(2), \dots, Y(201)$ .

**Оператор DIM.** В предыдущих разделах мы уже говорили о том, как на языке бейсик происходит работа с массивами. Однако до сих пор мы оставили в стороне один важный аспект этой работы. Дело в том, что при переводе программы с алгоритмического языка на внутренний язык машины необходимо знать, какой объем машинной памяти нужно отвести под тот или иной массив, встречающийся в тексте программы. Эту информацию транслятор получает из описания массивов. Поэтому каждый массив в программе должен быть описан. Для этого служит оператор DIM (сокращение от слова DIMENSION — размер). Его общий вид:

$k \text{ DIM } m_1 (\text{гр. } m_1), m_2 (\text{гр. } m_2), \dots, m_n (\text{гр. } m_n)$

Здесь  $k$  — метка оператора; DIM — наименование оператора;  $m_1, m_2, \dots, m_n$  — идентификаторы массивов; гр.  $m_1, \text{гр. } m_2, \dots, \text{гр. } m_n$  — границы массивов соответственно  $m_1, m_2, \dots, m_n$ , причем указываются только верхние границы массивов, так как нижние границы считаются равными единице.

Если массив  $m_i$  — одномерный, то гр.  $m_i$  — одно целое число, задающее количество элементов этого массива. Например, запись вида

10 DIM A(20)



говорит о том, что в программе будет использован массив, его идентификатор A и массив состоит из двадцати элементов. Если же массив  $m_i$  — двумерный (матрица), то нужно указать длины и строки и столбца матрицы. Поэтому запись гр.  $m_i$  будет состоять из двух целых чисел, разделенных запятой. Первое число задает длину строки, или, что то же самое, максимальное значение первого индекса массива, второе — длину столбца, или максимальное значение второго индекса массива.

Так, например, запись

20 DIM A (3, 4)

говорит о том, что в программе будет использован массив, его имя A, он двумерный, максимальное значение первого индекса — 3, второго — 4. Следовательно, всего элементов в этом массиве  $3 \times 4 = 12$ . И транслятор будет знать, какой объем памяти надо отвести под массив A.

Подчеркнем, что описание массива обязательно должно предшествовать его использованию в программе. В противном случае будет зафиксирована ошибка.

**Функции.** При решении задач очень часто используют элементарные функции, такие, как  $\sin x$ ,  $\cos x$ ,  $\ln x$ ,  $e^x$  и т. д.

При программировании задач на бейсике программисту нет необходимости самому писать программы для вычисления значений этих функций. К его услугам имеется набор элементарных функций и простой способ обращения к вычислению их значений. Для этого достаточно написать стандартное имя такой функции и за ним в круглых скобках ее аргумент.

Таблица 5.1

Элементарные функции

Обозначение функции на бейсике	Пояснения
SIN(X)	$\sin x$ , $x$ задается в радианах
COS(X)	$\cos x$ , $x$ задается в радианах
ATN(X)	$\operatorname{arctg} x$
SQR(X)	$\sqrt{x}$
SGN(X)	$\operatorname{sign} x = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$
EXP(X)	$e^x$
LOG(X)	$\ln x$
ABS(X)	$ x $ — вычисление абсолютной величины
INT (X)	вычисление целой части аргумента $x$

В табл. 5.1 приведены некоторые из элементарных функций, предоставляемых бейсиком.

Заметим, что для вычисления значений функций  $y = \text{sign } x$  и  $y = |x|$  мы составили свои программы в разделе, где изучали оператор IF. Там мы это делали просто для демонстрации возможностей оператора IF.

Можно было бы расширить перечень таких функций, которые часто называют стандартными. Но и этих достаточно, чтобы вычислить любую тригонометрическую функцию. Для вычисления логарифма по любому основанию достаточно воспользоваться известной формулой

$$\log_b a = \ln a / \ln b.$$

Поясним смысл функции INT(X). Она вычисляет значение наибольшего целого числа, не превосходящего X. Например,

$$\text{INT}(5.6) = 5$$

$$\text{INT}(-3.2) = -4$$

Как видно из примеров, результат вовсе не обязательно совпадает с числом, которое мы бы получили, если бы просто округляли аргумент X.

Если же нам понадобится округлить число X до ближайшего целого, то достаточно написать  $\text{INT}(X + 0.5)$ . Например, округлим число 5,6:

$$\text{INT}(5.6 + 0.5)$$

Вычислится значение функции INT (6.1), результатом которой будет число 6, что нам и нужно. Может случиться, что нам в программе придется неоднократно вычислять значение функции, задаваемой некоторой формулой, зависящей от нескольких переменных. При каждом обращении к функции значения аргументов могут меняться. Писать для каждого нового набора аргументов новый фрагмент программы — занятие малопродуктивное. Поэтому обычно поступают так: пишут выражение, соответствующее этой формуле, на языке бейсик лишь один раз, присваивают ему определенное имя, а затем обращаются к вычислению по этой формуле точно так же, как и к стандартной функции, указав только ее имя и набор аргументов. Для этой цели используют оператор DEF (сокращенно от DEFINITION — определение).

Его общий вид:

$$k \text{ DEF FNG}(x_1, x_2, \dots, x_n) = e$$

Здесь  $k$  — метка оператора; DEF — наименование оператора; FNG — имя нестандартной функции, причем первые два символа — FN пишутся у всех нестандартных функций, а G отличает данную функцию от других;  $x_1, x_2, \dots, x_n$  — список формальных аргументов;  $e$  — выражение.

Таким образом, в имени любой нестандартной функции две буквы всегда одни и те же — FN. А имя любой функции, как стандартной, так и нестандартной, состоит из трех латинских букв. Поэтому нестандартных функций в программе может быть, не больше, чем букв в латинском алфавите, т. е. 26.

Формальными аргументами  $x_1, x_2, \dots, x_n$  могут быть только простые переменные. Они используются для обозначения аргументов в формуле. При обращении к нестандартной функции эти формальные аргументы заменяются на фактические значения, для которых вычисляется функция.

Пусть при решении какой-то задачи нам нужно много раз пользоваться теоремой косинусов для вычисления длин сторон треугольников.

Мы можем описать формулу для вычисления лишь один раз, а обращаться к ней сколько угодно из разных мест программы. Для этого только будем каждый раз писать имя функции и указывать конкретные значения аргументов в формуле. Причем эти аргументы могут быть как числами, так и переменными, которым к моменту обращения к функции присвоены числовые значения.

Программа может выглядеть, например, так:

```
30 DEF FNA(X, Y, Z) = SQR (X↑2 + Y↑2 - 2*X*Y*cos(Z)
. . . . .
80 LET C = 0.5234
90 LET Y = 1.2
100 LET B = FNA(Y, 2.1, C)/2
. . . . .
150 IF B < FNA(Y/2, 1.5, 2*C - 1) THEN 210
```

Функция здесь описана в 30-й строке. От других описаний нестандартных функций ее отличает третья буква в имени функции — FNA, т. е. буква А. Формальными аргументами являются переменные X, Y и Z. Обращение к функции происходит дважды — в строке 100 и в строке 150. В 100-й строке фактические аргументы — переменные Y и C, которым перед этим обращением присвоены числовые значения (это присвоение могло произойти и где-то раньше, лишь бы к данному моменту Y и C имели какие-то числовые значения), и число 2.1.

В результате выполнения оператора присваивания в 100-й строке будет вычислено значение выражения

$$\sqrt{1.2^2 + 2.1^2 - 2 \cdot 1.2 \cdot 2.1 \cos 0.5234/2},$$

и это значение присвоится переменной B.

Фактических аргументов при обращении к функции должно быть столько же, сколько формальных в описании функции, и расположены они должны быть в соответствующем порядке.

Для обозначения формальных аргументов в описаниях функций могут использоваться любые идентификаторы, в том числе те, которые уже использованы или будут использованы в программе для других целей. Так, в нашем примере в описании функции используется формальный аргумент Y и этот же идентификатор используется в последующих операторах с другим смысловым значением.

Если в программе описано несколько стандартных функций, то для формальных аргументов разных функций можно использовать одинаковые имена переменных. Например,

```
10 DEF FNA(X, Y) = X↑2 + Y↑2
```

```
20 DEF FNB(X) = EXP(X)/2
```

Заметим еще, что входящее в описание функции выражение может содержать любые функции, как стандартные, так и описанные ранее нестандартные (кроме самой описываемой функции). Мы уже воспользовались стандартными функциями при описании примера с теоремой косинусов, а также в последнем примере (EXP(X)).

Приведем пример с применением нестандартной функции в выражении, входящем в описание другой функции:

```
20 DEF FNA(X) = ABS(SIN(X))
```

```
50 DEF FNB(X, Y) = FNA(X) + COS(Y)↑2 - 1
```

```
55 LET X = 2.5
```

```
60 LET Z = FNB(4.7124, 3.1416)*EXP(X)
```

По этой программе будет вычислено значение

$$z = (|\sin 4.7124| + \cos^2 3.1416 - 1) \cdot e^{2.5} \approx 12.1825.$$

Заметим, что обращение к функции не есть оператор. Оно всегда есть операнд в выражении при записи каких-либо операторов. Возможно, что все выражение, как, например, в строке 20 последнего примера, состоит только из обращения к функции. Оно используется либо в правой части оператора присваивания, либо в отношениях в операторах IF и WHILE, либо в описаниях функций.

**Подпрограммы.** Из предыдущего раздела мы знаем, что с помощью функции можно вычислить лишь одно числовое значение и лишь тогда, когда его можно задать одной формулой. Но, естественно, может быть множество случаев, когда нужно многократно выполнять одинаковые действия, которые либо нельзя записывать в виде одной формулы, а потребуются несколько операторов, либо эти действия должны давать в результате не одно значение, а массив числовых значений, например вектор. В таких случаях в бейсике используются подпрограммы. О подпрограммах мы уже говорили, когда рассказывали о приемах программирования на машинно-ориентированном языке. По существу, речь идет о том же понятии, но теперь мы попытаемся разъяснить, как оформляется подпрограмма на языке бейсик.

Подпрограмма—это специальным образом оформленная группа операторов. К ней можно обращаться из разных мест в программе. Для обращения к подпрограмме служит оператор GOSUB (SUB—сокращение от SUBROUTINE—подпрограмма), общий вид записи которого:

$k \text{ GOSUB } m$

Здесь  $k$ —метка оператора; GOSUB—наименование оператора;  $m$ —номер строки (метка) первого оператора подпрограммы. Оператор GOSUB несколько напоминает оператор GOTO. Отличие его от GOTO заключается в том, что он не только передает управление на оператор  $m$ , но и запоминает место, из которого произошло обращение. И как только подпрограмма закончит свою работу, произойдет возврат в то место основной программы, откуда было обращение, вернее, на тот оператор, который идет непосредственно за оператором обращения GOSUB.

Особенностью оформления последовательности операторов в подпрограмму является обязательное присутствие в подпрограмме оператора RETURN (возврат). Именно в результате работы этого оператора и происходит возврат из подпрограммы в основную программу.

В отличие от описания функций, подпрограммы в бейсике не имеют имен. Нег в подпрограммах и автоматической замены формальных аргументов на фактические. Программист, прежде чем обратиться к подпрограмме, сам должен задать для подпрограммы исходные значения. Должен он позаботиться и о том, чтобы результаты счета подпрограммы после окончания ее работы были присвоены соответствующим переменным. Как это происходит, мы поясним ниже на примере.

**Пример.** Составим программу для вычисления значения многочлена

$$P_{30}(x) = a_0x^{30} + a_1x^{29} + a_2x^{28} + \dots + a_{29}x + a_{30}$$

в точках  $u$ ,  $v$  и  $w$  так, чтобы получить  $l = P_{30}(u)$ ,  $m = P_{30}(v)$  и  $n = P_{30}(w)$ . Для переменных  $u$ ,  $v$ ,  $w$ ,  $l$ ,  $m$ ,  $n$ ,  $x$  введем соответственно обозначения  $U$ ,  $V$ ,  $W$ ,  $L$ ,  $M$ ,  $N$ ,  $X$ , массив коэффициентов многочлена  $a_0, a_1, \dots, a_{30}$  обозначим соответственно  $A(1), A(2), \dots, A(31)$ .

Основная программа:

```

100 LET X=U
110 GOSUB 1500
120 LET L=Y
130 LET X=V
140 GOSUB 1500
150 LET M=Y
160 LET X=W
170 GOSUB 1500
180 LET N=Y

```

Подпрограмма:

```
1500 LET Y = A(I)
1510 FOR I = 2 TO 31
1520 LET Y = Y * X + A(I)
1530 NEXT I
1540 RETURN
```

Подпрограмма вычисляет значение многочлена в некоторой точке X, результат присваивает переменной Y. Нам в действительности нужно вычислить значения многочлена для набора аргументов U, V и W, а результат присвоить соответственно переменным L, M и N. Основная программа как раз и задает подпрограмме информацию о том, для какого значения аргумента нужно вычислить многочлен.

Например, команда

```
100 LET X = U
```

присваивает переменной X значение первого аргумента. После этого происходит обращение к подпрограмме, которая вычисляет многочлен при этом значении аргумента. Результат в подпрограмме присваивает переменной Y, после чего с помощью оператора RETURN происходит возврат в основную программу, к 120-й строке. Там вычисленный результат присваивается переменной L, что и требовалось. Затем в 130-й строке основной программы переменной X присваивается новое значение V, и снова происходит обращение к подпрограмме. Там вычисляется значение многочлена для этого нового значения аргумента. Но результат в подпрограмме опять присваивается переменной Y. Возвращаемся в основную программу — в 150-ю строку, где результат вычисления присваивается переменной M. Наконец, переменной X присваивается значение W, и подпрограмма вычисляет многочлен для этого аргумента. После возвращения в основную программу результат присваивается переменной N.

При каждом обращении к подпрограмме происходит вычисление значения многочлена для нового значения аргумента. Переменная X в подпрограмме играет ту же роль, что формальные аргументы в описании функции. Но только при обращении к функции достаточно было просто перечислить в круглых скобках фактические аргументы. А здесь приходится перед каждым обращением к подпрограмме фактические аргументы присваивать переменным, играющим в подпрограмме роль формальных аргументов. В нашем примере такой формальный аргумент — X, а фактические — поочередно U, V и W. И каждый раз результат работы подпрограммы присваивается одной и той же переменной Y. Поэтому в основной программе нужно этот результат после работы подпрограммы присваивать именно тем переменным, которые указаны в условии задачи, т. е. соответственно L, M и N.

Если в подпрограмме есть разветвления вычислительного процесса, то может оказаться, что необходимо иметь возможность вернуться в основную программу после работы каждой из ветвей подпрограммы. Тогда в каждой такой ветви нужно поставить оператор возврата RETURN.

Как подпрограммы, так и операторы обращения к ним могут располагаться в любом месте программы. Внутри подпрограммы могут быть обращения к другим подпрограммам. Происходит это обращение точно таким же образом, как мы описывали выше.

Кроме подпрограмм, которые составляет сам программист, он может использовать в своей программе и стандартные подпрограммы. Стандартные подпрограммы входят в состав математического обеспечения ЭВМ, и для обращения к ним нужно только задать фактические аргументы и выбрать результаты.

**Оператор REM.** Как мы убедились, программы на языке бейсик получаются более компактными и наглядными, чем программы на машинно-ориентированном языке. Но тем не менее и их бывает полезно снабдить комментариями, которые могли бы облегчить чтение программы.

Часто бывает удобно разбить всю задачу на отдельные куски и каждый кусок как-то выделить, озаглавить. Хорошо бы также снабдить пояснениями сложные участки программы. Для этого в языке бейсик существует оператор REM (REM — сокращение от REMARK — замечание). Общий вид оператора:

$k$  REM <набор любых символов алфавита бейсика>

Операторы REM — неисполняемые, т. е. они не влияют на ход работы программы. Мы могли бы, например, программу на с. 194 снабдить заголовком и комментариями. Тогда она бы приняла вид

1500 REM ВЫЧИСЛЕНИЕ ЗНАЧЕНИЯ МНОГОЧЛЕНА

1510 LET Y = A(I)

1520 REM ЗАГОЛОВОК ЦИКЛА

1530 FOR I = 2 TO 31

1540 REM ТЕЛО ЦИКЛА

1550 LET Y = Y \* X + A(I)

1560 NEXT I

1570 REM КОНЕЦ ПРОГРАММЫ

**Операторы ввода и вывода информации.** Для того чтобы ЭВМ могла выполнить работу по обработке какой-либо информации, эта информация должна быть предварительно помещена в память машины. Иначе говоря, некоторым переменным и массивам нужно задать их начальные значения. Ранее уже говорилось о том, как это делается с помощью оператора присваивания LET и с помощью оператора чтения READ информации из блока данных. В ходе работы программы информацию в ЭВМ можно также вводить и с внешних

устройств. Эта информация хранится на перфокартах, перфоленте, магнитных лентах, магнитных барабанах, магнитных дисках. Чтобы в нужный момент информация с этих носителей была введена в ЭВМ, в программу включают специальные операторы ввода. Таких операторов может быть сколько угодно, и размещаться они могут в любом месте программы.

Язык бейсик наиболее удобен для работы с машиной в диалоговом режиме, когда вся информация в ЭВМ поступает с терминала. В этом случае каждая строка информации набирается на клавиатуре и тут же высвечивается на экране дисплея. Для перехода к набору следующей строки необходимо вслед за оператором текущей строки набрать на клавиатуре специальный управляющий сигнал. На разных ЭВМ он может называться по-разному, например, «возврат каретки» или «ввод». Обычно для этого достаточно нажать соответствующую клавишу.

С помощью терминала пользователь редактирует программу, т. е. вносит в нее все необходимые исправления, управляет ходом выполнения программы. На экран дисплея или на печатающее устройство (если оно есть) выводится результат работы программы. Это могут быть числа, таблицы, графики, рисунки, тексты.

Более подробно о работе с ЭВМ в диалоговом режиме мы поговорим позже. А пока рассмотрим, как вводятся данные с терминала по запросу программ.

Для ввода данных с терминала служит оператор INPUT (ввести). Его общий вид:

`k INPUT <список переменных>`

Здесь `k` — метка оператора; INPUT — наименование оператора; `<список переменных>` — последовательность переменных, разделенных запятыми.

Работает этот оператор следующим образом. Как только при выполнении программы машина встречает оператор ввода INPUT, так сразу же на экране дисплея появляется вопросительный знак (на некоторых ЭВМ вместо вопросительного знака высвечивается строка с оператором INPUT). Машина приостанавливает свою работу, и во время этой паузы пользователь должен набрать на клавиатуре числа для ввода, разделив их запятыми. А затем нужно нажать клавишу «Ввод». Тогда первое число станет значением первой переменной в списке переменных оператора ввода, второе число — значением второй переменной и т. д. То есть все происходит так же, как при работе оператора READ при чтении информации из блока данных. Необходимо только следить за тем, чтобы количество переменных в списке переменных совпадало с количеством чисел, набираемых для ввода. Если введено недостаточное количество данных или наоборот, их введено слишком много, машина выдает на экран сообщение о соответствующей ошибке. В этом случае необходимо повторить ввод,



т.е. заново набрать вводимые числа и нажать клавишу «Ввод». Если при наборе опять получилось, что набрали не то количество данных, которое требовалось командой INPUT, то вновь на экране дисплея появится сообщение об ошибке. Придется заново набрать числа для ввода. И так далее. Наконец, числа ввелись правильно. После этого машина приступит к выполнению оператора, следующего за INPUT.

Пусть, например, в каком-то месте программы требуется ввести в память ЭВМ числа 1.5; —0.004; 125 и присвоить эти значения соответственно переменным X, Y и Z.

Этот фрагмент программы может выглядеть следующим образом:

```
.....  
120 INPUT X, Y, Z  
130 LET A=X*Y-1  
.....
```

Встретив оператор INPUT в 120-й строке, машина высветит на экране дисплея вопросительный знак (или же строку 120 INPUT X, Y, Z) и будет ждать, пока пользователь наберет на клавиатуре какие-нибудь три числа. Если набрать последовательность чисел

1.5; —0.4E—2; 0.125E3

и нажать клавишу «Ввод», то произойдет присваивание этих числовых значений соответственно переменным X, Y и Z.

Если по ошибке пользователь наберет только два числа, например —1.5; —0.4E—2, то машина даст знать об ошибке либо сообщением

### ПОВТОРИТЕ ВВОД

либо

### ОШИБКА XX В СТРОКЕ 120

По коду каждой ошибки пользователь с помощью специальной таблицы может определять ее характер (знаками XX мы обозначили код данной ошибки).

В этом случае пользователь должен заново набрать на клавиатуре три числа и нажать клавишу «Ввод».

Может случиться, что пользователь набрал на клавиатуре большее количество чисел, чем требовалось оператором INPUT, например,

1.5, 1.5, —0.4E—2, 0.125E3

Тогда машина опять-таки выведет сообщение об ошибке (возможно, с указанием строки и кода ошибки) и потребует повторить ввод (в некоторых ЭВМ будет введено столько чисел, сколько требует оператор INPUT, а остальные проигнорируются).

Еще раз подчеркнем, что операторов INPUT может быть сколько угодно много и размещаться они могут в любом месте программы.

Вывод информации из оперативного запоминающего устройства ЭВМ может осуществляться на перфокарты, перфоленту, магнитные

ленты, магнитные барабаны, магнитные диски, алфавитно-цифровое печатающее устройство и на экран дисплея.

Мы ограничимся случаем, когда вывод результатов вычислений происходит на экран дисплея. Для этого используется оператор PRINT (печатать). Его общая форма:

$k$  PRINT <выводной список>

где  $k$  — метка оператора; PRINT — наименование оператора; <выводной список> — последовательность элементов выводного списка, разделенного либо запятой, либо точкой с запятой. Элементами выводного списка могут быть:

- а) числа;
- б) переменные,
- в) выражения,
- г) тексты,
- д) функция TAB(X).

Работает оператор PRINT следующим образом. Если нужно вывести на экран какие-либо числа, то их нужно просто выписать в выводном списке. В этом же списке можно указать имена переменных, значения которых мы хотим увидеть на экране.

Пусть, например, к моменту выполнения оператора

20 PRINT 24, A, X, 12

переменные A и X имели соответственно значения 1 и 5. Тогда после выполнения оператора PRINT на экране появятся числа 24, 1, 5 и 12 (мы пока не говорим о том, как эти числа будут расположены на экране).

В выводном списке могут стоять выражения и функции. В этом случае сначала будут вычислены их значения, а затем эти значения появятся на экране, т. е. возможны, например, записи такого вида:

30 PRINT A, 10, B, EXP(X), SQR(A↑2 + B↑2)

На экран можно выводить и произвольные тексты, состоящие из символов алфавита языка бейсик, в частности букв русского алфавита (если они входят в алфавит используемой версии языка). Этот текст заключается в кавычки и не должен содержать внутри себя кавычек. Например, в результате работы операторов

10 LET X = EXP(1)

20 PRINT "ЗНАЧЕНИЕ X ="; X

на экране появится текст

ЗНАЧЕНИЕ X = 2.71828

Расположение выводимой информации на экране в различных ЭВМ разное. Если на каждое выводимое число отводить 15 позиций, то на экране обычно можно разместить в каждой строке 4—5 чисел в зависимости от максимальной длины строки.

Предположим, что в строке не менее 60 позиций. Тогда ее можно разбить на зоны по 15 позиций и в каждой зоне записать одно число. Выше говорилось о том, что элементы выводного списка разделяются запятой или точкой с запятой. Так вот, если они разделяются запятой, то числа расположатся так, как мы только что сказали, т. е. по одному числу в зоне. Например, оператор

```
50 PRINT A, B, SQR(A↑2+B↑2)
```

выводит на печать три числа следующим образом:

значение A	значение B	значение $\sqrt{A^2+B^2}$	
1 поз.	16 поз.	31 поз.	46 поз.

Числа могут выводиться с фиксированной и плавающей точкой. Это зависит от их величины и от типа ЭВМ. Если в выводном списке окажется более четырех величин, то первые четыре печатаются в 1-й строке, следующие четыре — во 2-й и т. д.

Например, в результате выполнения оператора

```
30 PRINT 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

на экране высветится

```
1  2  3  4
5  6  7  8
9 10
```

Если мы хотим, чтобы выводимая информация размещалась на экране более плотно, то мы должны употребить в качестве разделителя в выводном списке вместо запятой точку с запятой. Тогда очередное число будет печататься не в следующей зоне, а через два пробела (или какое-либо другое количество пробелов). Например, оператор

```
10 PRINT 1; 2; 3; 4; 5; 6
```

приведет к выводу на экран

```
1 2 3 4 5 6
```

Опять-таки, если в выводном списке столько значений величин, что они не умещаются в одной строке, то числа продолжают выводиться на следующих строках.

Можно выводимые величины записать не в одном, а в нескольких подряд идущих операторах PRINT. Причем, если выводной список оператора PRINT заканчивается запятой, то следующий за ним оператор PRINT выводит информацию так, как будто величины его выводного списка являются продолжением выводного списка первого оператора.

Например, результат выполнения операторов

```
10 PRINT X, Y, Z,  
20 PRINT A, EXP(X)
```

тот же, что у оператора

```
30 PRINT X, Y, Z, A, EXP(X)
```

В этом случае значение *A* напечатается в 1-й строке, а значение *EXP(X)* — во второй. Отсутствие запятой или точки с запятой после выводного списка оператора *PRINT* приводит к тому, что вывод очередных значений будет происходить с новой строки.

Поэтому, если нам нужно пропустить несколько строк при выводе информации, мы можем поставить соответствующее количество операторов *PRINT* с пустым выводным списком. В этом случае роль каждого такого оператора будет заключаться только в переходе к следующей строке при очередном выводе.

Так, последовательность операторов

```
30 PRINT "ВЫДАЧА ЧИСЕЛ"  
60 PRINT  
70 PRINT 1, 5;  
80 PRINT 6  
90 PRINT 9, 10; 12  
100 PRINT  
120 PRINT  
130 PRINT "КОНЕЦ ВЫДАЧИ"
```

выведется на печать:

ВЫДАЧА ЧИСЕЛ

1	5	6
9	10	12

КОНЕЦ ВЫДАЧИ

Большое удобство при выдаче информации представляет использование функции *TAB(X)*. Она позволяет любой элемент выводного списка располагать в том месте строки, где нам захочется. В этой функции аргумент *X* указывает позицию, начиная с которой будет располагаться в строке очередной элемент выводного списка. Номер позиции является целым числом. Аргумент функции *TAB(X)* — выражение, вовсе не обязанное иметь целое значение. Поэтому после вычисления этого выражения берется его целая часть, и если она больше номера текущей позиции, то эта целая часть и задает номер позиции следующего элемента выводного списка. В противном случае номер позиции не меняется. Например, оператор

```
20 PRINT A; TAB(20); B; TAB(32 + 10/3); A + B
```

приведет к тому, что значение *A* будет печататься с первой позиции, *B* — с 20-й позиции, а значение *A + B* — с 35-й позиции. Между тем

оператор

30 PRINT A, B, A+B

разместил бы значения A, B и A+B соответственно с позиций — 1-й, 16-й и 31-й, т. е. в 1-й, 2-й и 3-й зонах строки.

Если вдруг окажется, что аргумент X функции TAB(X) дает значение номера позиции, превышающего общее количество позиций в строке, то вывод продолжается с начала следующей строки. Заметим, что на действие функции TAB(X) не влияет, какой разделитель — запятая или точка с запятой стоит после нее.

Функцию TAB(X) особенно удобно использовать при построении приближенных точечных графиков каких-либо функций. Для такого построения выбирают какой-либо подходящий символ из набора символов клавиатуры (чаще всего звездочку (\*) либо символ знака плюс (+)). Этот символ размещается в позиции экрана, который соответствует значению ординаты функции. Обычно мы привыкли, что графики рисуются так, чтобы ось X (ось аргументов) была расположена горизонтально, а ось Y (ось ординат значений функции) — вертикально. Поскольку на экране и печатающем устройстве информация выводится по строкам сверху вниз, то и графики выглядят здесь довольно своеобразно: ось X располагается сверху вниз, ось Y — вдоль строки.

Для построения графиков функции удобно использовать оператор вида

`PRINT TAB(X); "<символ>"`

Здесь X указывает номер позиции, в которой будет напечатан <символ> — любой символ, имеющийся в наборе символов клавиатуры терминала.

Естественно, при построении графика какой-либо функции надо стремиться к оптимальному размещению графика на плоскости экрана. Так, если функция симметрична относительно оси абсцисс, как, например,  $y = \sin x$  или  $y = \cos x$ , то желательно, чтобы ось абсцисс шла по середине экрана. Если же функция только положительна, как, например,  $y = e^x$ , или, наоборот, только отрицательна, как, например,  $y = -|x|$ , то ось абсцисс должна идти по краю экрана. В этих случаях графики функций могут занимать всю площадь экрана. Если же мы всегда размещали бы ось абсцисс в середине экрана, то такая функция, как  $y = e^x$ , могла бы быть нарисована только на половине экрана, а именно той ее части, которая соответствует положительному значению оси ординат, а вторая половина экрана была бы пустой.

Поэтому прежде, чем рисовать график какой-либо конкретной функции  $y = f(x)$ , необходимо посмотреть, каковы ее минимальное и максимальное значения на том интервале значений аргумента функции, на котором мы хотим ее строить.

Будем считать, что в каждой строке экрана 60 позиций — с 1-й по 60-ю. Для оптимального использования площади экрана надо сделать так, чтобы точки, соответствующие минимальным значениям функции, располагались в левом краю экрана, т. е. в первых позициях, максимальным — в правом, т. е. в районе 60-й позиции.

Ось абсцисс у нас будет располагаться по вертикали, а сам график как бы положен набок, т. е. ось ординат пойдет слева направо. График функции  $y = \sin x$  должен выглядеть так, как на рис. 5.12.

Зная количество позиций в строке, а также минимальное и максимальное значения функции  $y = f(x)$ , мы можем выбрать масштаб для построения графика. Если максимальное значение функции равно  $M$ , а минимальное  $N$ , и в строке 60 позиций, то в качестве единицы масштаба можно взять  $H = 50/(M - N)$  (мы взяли для определения масштаба не 60, а 50 позиций, чтобы точки графика не выходили на самые края экрана). Например, для функции  $y = x^2$  на отрезке  $x \in [0; 5]$  единица масштаба будет равна  $50/25 = 2$ , так как  $M = 25$ , а  $N = 0$ . Если  $Y$  — текущее значение функции, точки графика которой мы хотим вывести на экран, то для определения позиции (ординаты) точки на экране можно воспользоваться стандартной функцией  $INT(X)$  — вычисление целой части аргумента  $X$ :

$$INT((Y - N) * H + 5)$$

(5 позиций мы добавили к аргументу, чтобы график функции не выходил на левый край экрана).

Действительно, для  $y = x^2$  значениями функции  $INT$ , например, будут

$$\begin{aligned} \text{при } y = 0 \quad & INT((0 - 0) * 2 + 5) = INT(5) = 5 \\ \text{при } y = 2 \quad & INT((4 - 0) * 2 + 5) = INT(13) = 13 \\ \text{при } y = 3,5 \quad & INT((12,25 - 0) * 2 + 5) = INT(29,5) = 29 \\ \text{при } y = 5 \quad & INT((25 - 0) * 2 + 5) = INT(55) = 55 \end{aligned}$$

Мы видим, что для построения графика функции будут использованы практически все позиции экрана.

И теперь, чтобы вывести на экран какой-либо символ, соответствующий текущему значению функции  $Y$ , промасштабированному с помощью оператора  $INT$ , достаточно выполнить оператор

```
PRINT TAB (INT((Y - N) * H + 5)); "СИМВОЛ"
```

Так, если мы будем рисовать ось абсцисс с помощью символа «!», то можно записать оператор печати в виде

```
PRINT TAB(INT((0 - N) * H + 5)); "I"
```

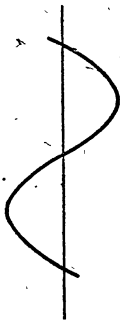


Рис. 5.12

Этот оператор должен выполняться в цикле так, чтобы каждый раз символ «!» печатался в следующей строке.

Например, для функции  $y=x^2$  ось абсцисс можно строить с помощью оператора PRINT TAB(INT((0-0)\*N+5)); "!" или же просто

```
PRINT TAB(5); "!"
```

Символ «!» будет печататься в 5-й позиции, так как аргумент функции TAB равен 5.

Для иллюстрации сказанного выше составим программу построения графика функции  $y=\sin x$  на отрезке  $x \in [-2\pi, 2\pi]$  с шагом изменения аргумента  $\Delta x = \pi/16$ . Ось абсцисс будем рисовать символом «!», а точки графика — символом «\*».

Поскольку минимальное значение функции  $y=\sin x$  на заданном интервале равно  $N=-1$ , а максимальное  $M=1$ , то масштаб имеет значение  $50/(1-(-1))=25$ . Следовательно, ось абсцисс, определяемая уравнением  $y=0$ , пройдет по 30-й позиции экрана, так как

$$\text{INT}((0-(-1))*25+5) = \text{INT}(30) = 30$$

Минимальное значение функции  $y=\sin x$  будет находиться в 5-й позиции ( $\text{INT}((-1-(-1))*25+5) = \text{INT}(5) = 5$ ), а максимальное — в 55-й ( $\text{INT}((1-(-1))*25+5) = \text{INT}(55) = 55$ ).

Итак, составим программу на языке бейсик, которая вычислит таблицу значений  $y_i = \sin x_i$  на отрезке  $x \in [-2\pi, 2\pi]$  с шагом  $\Delta x = \pi/16$ , выведет на экран эту таблицу и график функции.

Заметим, что на экране это будет не непрерывная линия, а отдельные точки графика:

```
20 REM ПОСТРОЕНИЕ ГРАФИКА И ТАБЛИЦЫ Y=SIN X
25 DIM X(65), Y(65)
30 LET P=3.1416
35 LET N=-1
40 LET M=1
45 LET H=50/(M-N)
50 PRINT "ГРАФИК ФУНКЦИЙ Y=SIN X"
60 LET X(1)=-2*P
65 LET I=1
70 LET Y(I)=SIN(X(I))
80 IF Y(I) < 0 THEN 110
90 PRINT TAB(INT(0-N)*H+5)); "!"
100 PRINT TAB(INT((Y(I)-N)*H+5)); "*"
105 GOTO 120
110 PRINT TAB(INT((Y(I)-N)*H+5)); "*";
115 PRINT TAB(INT((0-N)*H+5)); "!"
120 LET I=I+1
130 LET X(I)=X(I-1)+P/16
```

```

140 IF I ≤ 65 THEN 70
150 REM ПЕЧАТЬ ТАБЛИЦЫ
155 PRINT "ТАБЛИЦА ЗНАЧЕНИЙ  $Y = \sin(X)$ "
160 FOR I=1 TO 65
170 PRINT X(I), Y(I)
180 NEXT I
190 END

```

Прокомментируем программу. В программе два цикла — один для вычисления таблицы значений функции  $y = \sin x$  и вывода на экран графика функции, другой для вывода на экран таблицы. Первый цикл осуществляется операторами 70—140, второй — 160—180. Ось абсцисс рисуется операторами 90 и 115, точка графика — операторами 100 и 110. Отсутствие после каждого из этих операторов знаков препинания говорит о том, что очередной символ появится в следующей строке экрана.

Можно несколько улучшить программу, если заметить, что значения некоторых одинаковых выражений вычисляются многократно. Например, в операторах 90 и 115 аргумент функции TAB всегда один и тот же:

```
INT((0—N)*H+5)
```

Поэтому его вычисление разумно было бы вынести за пределы цикла и, вычислив один раз, присвоить полученное значение некоторой переменной, например L, которую затем и использовать при выполнении операторов 90 и 115. Аналогично можно вычислить вне цикла значение  $P/16$ , присвоить его переменной P1. Тогда вне цикла появятся дополнительные операторы

```

52 LET L = INT((0—N)*H+5)
57 LET P1 = P/16

```

Но они выполняются лишь по одному разу. Зато внутри цикла уже можно будет при каждом повторении цикла не вычислять эти значения заново, если исправить операторы 90, 115 и 130 следующим образом:

```

90 PRINT TAB(L); "I";
115 PRINT TAB(L); "I"
130 LET X(I) = X(I—1) + P1

```

Эта программа представляет собой законченный продукт. Ее можно ввести в память ЭВМ и, выполнив ее, получить на экране дисплея результат — график функции  $y = \sin x$  и таблицу ее значений.

Наша программа завершается оператором END (конец). В принципе это не обязательно, так как после выполнения всех строк программы работа программы все равно закончится.

Программу для построения графика функции  $y = \sin x$  можно сде-



лать еще более простой, если воспользоваться оператором цикла

```
10 REM ИВАНОВ ГР1 ЗАДАНИЕ 1
20 PRINT "ГРАФИК ФУНКЦИИ Y=SIN X"
30 FOR X=-6.2831853 TO 6.2831853 STEP 0.1963495
40 PRINT TAB (30+20*SIN(X)); "*"
50 NEXT X
60 END
```

Здесь на отрезке  $[-2\pi, 2\pi]$  с шагом  $\pi/16$  строится график функции  $y = \sin x$ . Символ «\*» ставится в позицию, номер которой определяется целой частью значения:  $30 + 20 \times \sin x$ . Первое слагаемое показывает, где должна находиться ось абсцисс (но сама ось абсцисс не рисуется), а второе задает смещение точки графика относительно оси абсцисс. Например, это смещение будет равно

$$\text{при } x = \pi/4 \quad [20 \sin \pi/4] = [20 \cdot \sqrt{2}/2] = 14,$$

$$\text{при } x = \pi/2 \quad [20 \sin \pi/2] = 20,$$

$$\text{при } x = -\pi/2 \quad [20 \sin (-\pi/2)] = -20.$$

Таким образом, график займет по горизонтали 40 позиций — по 20 влево и вправо от 30-й позиции.

## § 5.4. Диалог «Человек — ЭВМ»

Одним из наиболее распространенных способов работы с ЭВМ является работа в диалоговом режиме.

Перед пользователем терминал. Вся информация, которую необходимо ввести в память ЭВМ, набирается на клавиатуре и тут же высвечивается на экране. Пользователь имеет возможность ее проверить и, в случае необходимости, исправить. Если в программе есть синтаксические ошибки, сообщение о них выдается на экран. После того как программа отредактирована, она переводится транслятором на внутренний язык машины и выполняется.

Во время счета может произойти останов в результате какой-либо ошибки. Машина тут же сообщит о причине останова.

Пользователь имеет возможность все время следить за ходом решения задачи, вмешиваться в него, внося какие-либо исправления, задавать машине исходные данные для счета, останавливать вычисления в нужный момент, чтобы посмотреть промежуточные результаты.

Все эти возможности пользователю предоставляет совокупность технических средств и математического обеспечения ЭВМ. В состав математического обеспечения ЭВМ входят различные алгоритмические языки, в частности, для диалогового режима одним из наиболее удобных языков является бейсик.

Для обработки записи, сделанной на языке бейсик, используется интерпретирующая или компилирующая система (см. с. 170), называемая бейсик-системой (но не надо путать различные понятия: язык

бейсик и бейсик-система). Каждая машина может иметь свою бейсик-систему. Но все они, отличаясь в деталях, должны предоставлять указанные выше возможности

Для работы с программой, записанной на алгоритмическом языке бейсик, в бейсик-системе используются специальные так называемые системные команды (их еще называют директивами). В отличие от обычных операторов в программе пользователя, они не имеют номера строки и выполняются сразу же после их ввода. Эти команды в различных типах машин и различных бейсик-системах могут отличаться как по форме, так и по содержанию. С их помощью осуществляется ввод информации в память ЭВМ, вывод всей информации или какой-либо ее части, редактирование программы, т. е. замена, вставка или вычеркивание символов в программе, запуск программы на трансляцию и выполнение.

Начинать работу на машине можно только после того, как на экране появится слово READY (готов). После этого на клавиатуре терминала набирают программу. Вслед за программой вводится системная команда (директива) RUN (пуск), которая является сигналом для начала трансляции вводимой программы. Если в программе не обнаружатся ошибки, то сразу же после трансляции начнет выполняться программа.

Таким образом, программа на языке бейсик с директивой RUN может выглядеть на экране, например, так:

```
10 LET X=2
20 LET Y=SQR(X)
30 PRINT "X="; X; "Y="; Y
40 END
RUN
```

Напомним, что при наборе программы, для перехода к следующей строке необходимо нажать клавишу «возврат каретки».

После набора директивы RUN программа оттранслируется; выполняется и на экране появится

```
X=2 Y=0,14142E1
```

На экране дисплея высвечивается как информация, которую мы набираем на клавиатуре, так и сообщения, выдаваемые машиной. Для того чтобы их не путать, мы сообщения, поступившие из машины будем выделять жирным шрифтом. Например,

```
READY
10 INPUT A, B
20 LET X=A/B
30 PRINT "X="; X
40 END
RUN
?
```

## ПОВТОРИТЕ ВВОД

4,2

$X=2$

В этой программе после трансляции потребовался ввод двух чисел А и В. Знак вопроса означает, что машина требует ввода (в некоторых ЭВМ вместо знака вопроса может высветиться строка с оператором ввода, т. е. в данном случае — INPUT A,B).

Мы по ошибке вместо двух чисел набрали лишь одно. Поэтому машина тут же потребовала повторить ввод. Второй раз мы вводимые данные набрали правильно, и машина перешла к следующему оператору. В результате работы программы на экране появится результат:  $X=2$ .

Если при вводе программы обнаружилась в какой-то строке ошибка, то эту строку надо целиком набрать заново. Ошибка может обнаружиться и после подачи директивы RUN. Тогда работа машины приостановится, и нужно заменить неправильные операторы на новые, исправленные операторы. Для того чтобы заменить какой-либо оператор, достаточно набрать на клавиатуре новый оператор, снабдив его тем же номером, что был у замененного оператора.

В процессе работы с программой может быть всегда полезным и еще ряд системных команд. Рассмотрим некоторые из них.

Директива DELETE (стереть). Эта директива (иногда она называется SCR, CLEAR) служит для стирания строк программы. Если нужно стереть только одну строку, например, с номером 100, то пишут

DELETE 100

Если нужно стереть последовательность строк, например, все строки от 50-й до 200-й включительно, то пишут

DELETE 50, 200

Наконец, если нужно стереть всю программу, то пишут просто

DELETE

Директива LIST (распечатать). Директива LIST используется для вывода на экран текста программы. Если после редактирования программы мы хотим посмотреть на экране какой-либо фрагмент, то необходимо набрать директиву LIST с указанием номеров начальной и конечной строк выводимой на экран информации. Например, директива

LIST 35, 140

выведет на экран все строки от 35-й до 140-й включительно. Если нужно посмотреть лишь одну строку, то достаточно после директивы LIST указать ее номер, например,

LIST 110

и 110-я строка программы появится на экране. Наконец, по директиве LIST

выведется вся программа.

Мы уже упоминали, что машина, а точнее говоря, бейсик-система, в случае обнаружения ошибок в программе выдаст соответствующие сообщения. Эти сообщения имеют вид

### ОШИБКА А В СТРОКЕ В

где А — цифровой код ошибки, В — номер строки, в которой произошла ошибка.

Так, в примере на с. 217 вместо сообщения ПОВТОРИТЕ ВВОД в некоторых ЭВМ выдается сообщение

### ОШИБКА 121 В СТРОКЕ 10

Код 121 означает, что причиной возникновения ошибки является недостаточное количество данных, вводимых с помощью оператора INPUT. Аналогичным образом закодированы и другие возможные ошибки. Например, код ошибки, вызванной извлечением корня из отрицательного числа 126; если есть оператор NEXT и нет соответствующего оператора FOR, то код ошибки 24.

Перечень кодов ошибок всегда имеется в инструкции по применению бейсик-системы.

**Отладка программы.** Пусть программа на языке бейсик, реализующая алгоритм решения некоторой задачи, составлена.

Можно ли быть уверенным, что она не содержит ошибок? Разумеется, нет. Обычно лишь начинающие программисты всегда уверены в правильности своих программ. Среди опытных программистов бытует афоризм: «В каждой программе есть хотя бы одна ошибка». Естественно, речь идет не о слишком уж коротких и примитивных программах, хотя начинающий программист ошибается и в них. Процесс отыскания ошибок в программах называют отладкой. Как же находить ошибки в программах?

Ошибки могут быть разного характера — синтаксические, т. е. нарушены правила записи операторов или вводимых данных, или же ошибки в самой структуре алгоритма. Что касается синтаксических ошибок, то их помогает выявить транслятор. В процессе трансляции программы он в случае обнаружения такой ошибки выдает на экран соответствующее сообщение, указав код ошибки и место ее расположения.

После того как все синтаксические ошибки устранены, еще ни в коем случае нельзя быть уверенным, что программа составлена верно. Это подобно тому, как безошибочное с точки зрения орфографии школьное сочинение далеко не всегда правильно раскрывает его тему. Поэтому необходимо еще выявить и устранить все ошибки в самом алгоритме.

Если, не устранив такие ошибки, попробовать выполнить программу, то в лучшем случае машина остановится, встретив какое-либо несуразное действие, например деление на нуль или вычисление логарифма отрицательного числа. Тогда уже ясно, причину какой ошибки надо искать. Хуже, если программа выполнится и выведет результат, а он окажется совсем не тем, каким должен был получиться. Причем мы не всегда, к сожалению, можем знать заранее, каким, хотя бы приблизительно, должен быть результат. Поэтому, даже если в программе нет синтаксических ошибок, ее все равно обязательно надо отлаживать.

В любой программе есть, как правило, разветвления, переходы, циклические процессы. Необходимо проверить как логическую структуру программы, так и вычисления на так называемых линейных участках программы, т. е. там, где операторы выполняются последовательно один за другим в порядке их записи.

Для отладки линейных участков программ поступают следующим образом. Пусть на некотором линейном участке программы происходят вычисления по какой-то, быть может, достаточно сложной формуле. Причем в ходе решения задачи обращение к этому участку происходит многократно и для различных значений аргументов формулы. Чтобы проверить правильность этого участка программы, задают для этой формулы некоторый определенный набор аргументов, называемый тестовым. Просчитывают вручную (можно воспользоваться и микрокалькулятором) формулу с этими значениями аргументов. А затем выполняют на ЭВМ тот участок программы, который производит вычисления по этой формуле с теми же аргументами. Если результаты ручного счета и машинного не совпадают, то надо искать ошибку. Ошибка может быть как в том, так и в другом способах решения. Поэтому надо проверить и результаты ручного счета.

Кстати, если для ЭВМ безразлично, какие числа взяты в качестве аргументов, то для ручного счета это, конечно, совсем не безразлично. Часто удобно, например, брать целые числа — единицу, двойку и т. д. При этом надо смотреть за тем, чтобы выбранные значения аргументов не помешали выявить возможные ошибки. Например, если вычисления вести по формуле

$$y = (a - 2) (\sin x + e^x)^2 - be^{-x},$$

то, задав значение  $a$ , равное или очень близкое к 2, мы не заметим возможных ошибок при вычислении значения выражения, стоящего в скобках, т. е.  $(\sin x + e^x)^2$ , даже если ошибка будет очень грубой.

Итак, перепроверив результаты ручного счета, будем искать ошибку в программе. Прежде всего надо еще раз за разом внимательно посмотреть этот участок программы. Если ошибка не обнаружена, то надо вставить в программу операторы вывода промежуточных резуль-

татов. Сравнив машинные результаты с полученными вручную, мы увидели бы, где они не совпадают, и тем самым локализовали бы ошибку, нашли ее, а затем устранили.

Проверив, с помощью тестов все линейные участки программы, можно приступить к проверке логической структуры программы, т.е. правильности организации циклов, разветвлений, переходов.

Пусть, например, надо составить программу для решения квадратного уравнения

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

Программа должна находить корни уравнения для наборов коэффициентов, которые последовательно вводятся в память ЭВМ.

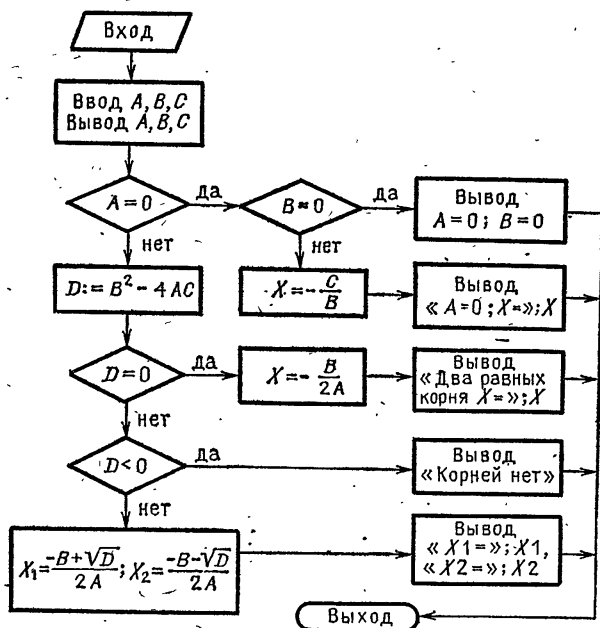


Рис. 5.13

Возможны различные случаи. Если  $a=0$ , то  $x = -c/b$  ( $b \neq 0$ ).

Если же  $a \neq 0$ , то возможны случаи, когда  $D = b^2 - 4ac < 0$  и  $D \geq 0$ . В первом случае действительных корней нет, а во втором — корни вычисляются по формуле

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

Блок-схема алгоритма представлена на рис. 5.13.

Приведем соответствующую программу на бейсике:

```

10 REM ВЫЧИСЛЕНИЕ КОРНЕЙ УРАВНЕНИЯ
20 INPUT A, B, C
30 REM ВЫВОД ДЛЯ ПРОВЕРКИ ВВОДА
40 PRINT A, B, C
50 IF A=0 THEN 210
60 LET D=B↑2-4*A*C
70 IF D=0 THEN 170
80 IF D<0 THEN 140
90 REM ВЫЧИСЛЕНИЕ КОРНЕЙ ПРИ D>0
100 LET X1=(-B+SQR(D))/(2*A)
110 LET X2=(-B-SQR(D))/(2*A)
120 PRINT "X1="; X1, "X2="; X2
130 GOTO 280
140 PRINT "КОРНЕЙ НЕТ"
150 GOTO 280
160 REM ВЫЧИСЛЕНИЕ КОРНЕЙ ПРИ D=0
170 LET X=-B/(2*A)
180 PRINT "ДВА РАВНЫХ КОРНЯ. X="; X
190 GOTO 280
200 REM ВЫЧИСЛЕНИЕ КОРНЯ ПРИ A=0
210 IF B=0 THEN 260
220 REM ВЫЧИСЛЕНИЕ КОРНЯ ПРИ B≠0
230 LET X=-C/B
240 PRINT "A=0, X="; X
250 GOTO 280
260 PRINT "A=0, B=0"
270 REM КОНЕЦ ПРОГРАММЫ
280 END

```

Для того чтобы проверить правильность работы программы, нужно задавать такие тестовые наборы аргументов, чтобы можно было быть уверенным, что каждая ветвь программы работает верно. То есть нужно проверить, всегда ли вычисления пойдут по правильному на-

Таблица 5.2

	A	B	C	Уравнение	Корни
I	0	0	4	$4=0$	$A=0, B=0$
II	0	2	4	$2x+4=0$	$x=-2$
III	1	-2	1	$x^2-2x+1=0$	$x_{1,2}=1$
IV	1	-1	-2	$x^2-x-2=0$	$x_1=2, x_2=-1$
V	1	-1	2	$x^2-x+2=0$	корней нет

правлению при заданном наборе аргументов. В нашем примере для этой проверки достаточно задать пять наборов аргументов (табл. 5.2).

По директиве RUN машина начнет выполнять программу.

Встретив команду ввода

20 INPUT A,B,C

машина потребует ввода чисел, т. е. выдаст на экран знак вопроса (или повторит запись команды ввода).

Мы введем первый набор данных. Машина должна выдать на экран информацию

$A=0, B=0$

и прекратить работу.

Если мы снова дадим директиву RUN, то машина снова начнет выполнять программу и опять потребует ввода чисел. Теперь мы введем второй набор исходных данных. На экране должно появиться

$A=0, X=-2$

Опять дадим директиву RUN и введем третий набор коэффициентов. На этот раз, поскольку  $A \neq 0$  и  $D=0$ , должны получить на экране

ДВА РАВНЫХ КОРНЯ.  $X=1$

Далее введем четвертый набор коэффициентов уравнения. Теперь уже, так как  $A \neq 0$  и  $D \neq 0$ , на экран выдаться

$X1=2 \quad X2=-1$

Наконец, при последнем наборе коэффициентов дискриминант отрицателен и на экране появится текст

КОРНЕЙ НЕТ

Если после ввода какого-либо из наборов данных окажется, что результат не совпадает с ожидаемым, надо искать ошибку в программе. Этими пятью наборами мы проверим, верно ли работает каждая из ветвей программы. Перечисленные выше наборы коэффициентов можно было вводить и в любом другом порядке.



## ГЛАВА 6

### ОБРАБОТКА ТЕКСТОВОЙ ИНФОРМАЦИИ

Мы уже упоминали ранее, что в качестве констант и значений переменных в бейсике могут использоваться не только числа, но и объекты другой природы, в частности тексты, геометрические фигуры.

Малые вычислительные машины, имеющие в своем составе экран, клавиатуру, печатающее устройство, удобно использовать в качестве своеобразной пишущей машинки для подготовки различного рода описаний, статей, документов и даже монографий. Сейчас в предисловии ко многим книгам научного содержания часто можно прочитать, что текст подготовлен с помощью ЭВМ такого-то типа.

Своеобразие использования микро-ЭВМ в этой работе заключается в том, что сначала весь набираемый на клавиатуре текст поступает в память ЭВМ. Этот текст можно извлечь из памяти, прочитать его на экране, и если необходимо, то исправить, дополнить и т. д. После этого его уже можно выдать на печатающее устройство.

Большая доля времени при работе с персональными ЭВМ уходит на их использование как раз в качестве инструмента обработки текстов. Если проанализировать, как мы работаем над рукописью, то можно усмотреть в этой работе ряд часто встречающихся операций, связанных с заменой одного слова на другое, изменением окончаний слов, вставкой новых слов или словосочетаний, выделением разделов и параграфов.

Если, например, записать в память ЭВМ фамилии всех учеников класса, названия дисциплин и те оценки, которые зафиксированы в журнале, то потом можно с помощью ЭВМ быстро получить информацию об успеваемости любого ученика, как только задана его фамилия. Для этого должна быть создана программа, отыскивающая по заданной фамилии относящиеся к ней сведения. Иными словами, возникает необходимость иметь средства разработки алгоритмов, работающих с текстами (словами, фразами, строками). Исполнитель такого рода алгоритмов должен уметь сравнивать тексты, преобразовывать тексты по самым различным правилам.

Учитывая то, что текстовая обработка приобретает все большее

значение при использовании ЭВМ в качестве инструмента-помощника, в языки программирования, которые вначале были рассчитаны только на обработку числовой информации, вводятся новые операции, позволяющие обрабатывать тексты. Этому процессу не избежал и язык бейсик.

Под текстом независимо от его содержания будем понимать последовательность любых алфавитно-цифровых символов — букв, цифр, служебных знаков. Каждый символ кодируется в машине, как мы знаем, последовательностью из восьми двоичных цифр, т. е. на каждый символ отводится один байт памяти. Тем самым в пределах одного байта есть возможность закодировать  $2^8 = 256$  различных символов. Причем каждому символу взаимно однозначно соответствует определенный набор двоичных цифр.

Значения, составленные из последовательностей символов, мы будем называть текстовыми. Иногда их еще называют литерными.

Так же, как и для чисел, в расширениях языка бейсик вводится понятие текстовой константы и текстовой переменной.

Текстовая константа — последовательность алфавитно-цифровых символов, заключенная в кавычки, например,

"МИР"

"РЕАСЕ"

"РЕЗУЛЬТАТ"

"ДЕСЯТЫЙ КЛАСС "А"

Заметим, что пробел в текстовой константе — это тоже символ.

Значением текстовой константы считаются символы, заключенные во внешние кавычки. Поэтому в последнем примере значением константы является

ДЕСЯТЫЙ КЛАСС "А"

По существу, мы уже встретились с текстовыми величинами, когда их использовали в операторе PRINT. Так, например, если к моменту выполнения этого оператора переменная X имела значение, равное 3.5, то в результате работы оператора

PRINT "X="; X

на экране высвечивается  $X = 3.5$ .

В бейсике допускается и использование текстовых переменных.

Идентификатор текстовой переменной, так же как и числовой, может состоять либо из прописной буквы латинского алфавита, либо из буквы с одной, стоящей за ней цифрой. Но, в отличие от числовой переменной, после буквы или буквы с цифрой здесь обязательно ставится специальный символ □ (солнышко); т. е. в качестве текстовой переменной можно использовать

A□, B□, ..., X□, Y□, Z□, A0□, B0□, ..., Z0□, A1□, B1□, ..., Z1□, A9□, B9□, ..., Z9□

Над текстовыми величинами, так же как и над числовыми, можно выполнять различные действия. Можно, например, с помощью оператора присваивания текстовым переменным присваивать различные значения. Например, с помощью оператора

```
LET A□ = "СКОРОСТЬ"
```

текстовой переменной с именем A□ присвоится значение СКОРОСТЬ.

В качестве правой части оператора присваивания может быть не только текстовая константа, но и текстовая переменная. Например,

```
LET X□ = Y□
```

Если значением переменной Y□ был набор символов СУММА, то после выполнения оператора присваивания это же значение будет иметь и переменная X□.

Различать между собой идентификаторы числовых переменных и текстовых необходимо не только для лучшего понимания работы алгоритмов, выраженных программой, но это важно и для интерпретации операторов. Например, действия очень похожих друг на друга операторов

```
LET A□ = "3.14"
```

и

```
LET A = 3.14
```

будут различны. В первом случае переменной A□ будет присвоено текстовое значение, состоящее из четырех символов: 3, ., 1, 4. Во втором случае переменная A примет значение, равное числу 3.14.

Текстовые величины наравне с числовыми могут использоваться в операторах ввода и вывода. Если, например, в программе встретятся операторы

```
60 INPUT X□: A, B  
70 PRINT X□: "="; B/A
```

то оператор ввода потребует ввода текстового значения переменной X□ и числовых значений переменных A и B. На экране дисплея высветится вопросительный знак: ?.

Пользователь, работающий за терминалом, должен с помощью клавиатуры набрать, например, следующие данные:

```
КОРЕНЬ УРАВНЕНИЯ, 4, 6
```

(вводимая текстовая величина не обязательно заключается в кавычки).

Получив эти данные, машина выполнит оператор вывода в 70-й строке, и на экране будет:

```
КОРЕНЬ УРАВНЕНИЯ=1.5
```

Заметим, что в операторе PRINT мы разделили переменные в списке вывода знаком «;», чтобы результаты выводились близко друг за другом.

Если бы мы поставили знак «», то между ними были бы слишком большие расстояния (каждое значение было бы в своей зоне в строке).

Количество символов в значении текстовой переменной может быть, вообще говоря, произвольным. Это количество называют длиной переменной. В бейсике длина значения текстовой переменной считается стандартной, если она не превышает шестнадцати, включая внутренние пробелы. В этом случае никаких специальных оговорок для действий над такими величинами делать не надо. Так мы и поступали до сих пор. Но если число символов в значении текстовой переменной может превысить 16, то необходимо прежде, чем эта переменная будет использована в программе, описать ее с помощью уже знакомого нам оператора DIM, указав в нем максимальную длину значения данной переменной. Максимальная длина текстовой переменной, задаваемой с помощью оператора DIM, может быть от 1 до 253. Если же ничего другого не оговорено в операторе DIM, то для каждой текстовой переменной или константы выделяются по 16 байт. Если это текстовое значение короче, то оно дополняется справа пробелами; если же длиннее, то символы, начиная с 17-го, отбрасываются. Пусть, например, нам известно, что текстовая переменная XQ может принимать значения, число символов в которых может достигать двадцати пяти. Тогда в результате работы операторов

```
30 DIM XQ 25
```

```
40 LET XQ="РЕЗУЛЬТАТ РЕШЕНИЯ ЗАДАЧИ"
```

```
50 PRINT XQ
```

на экране высветится

РЕЗУЛЬТАТ РЕШЕНИЯ ЗАДАЧИ

Здесь высветились 24 символа (пробел—тоже символ). Так как количество меньше максимального, то остальные символы справа (здесь это будет один символ) будут пробелами.

Если бы мы в этом случае не поставили оператор DIM, то высветились бы лишь первые 16 символов:

РЕЗУЛЬТАТ РЕШЕНИ

Аналогичная ситуация получилась бы, если бы выполнили операторы

```
20 LET AQ="КОРЕНЬ УРАВНЕНИЯ ="
```

```
30 PRINT AQ; 2.5
```

.....

Получили бы на экране

КОРЕНЬ УРАВНЕНИЯ 2.5

Знак равенства не появился—для него просто не хватило места, ибо он оказался семнадцатым символом в текстовой константе. Чтобы он не потерялся, достаточно было перед операторами 20, 30 вставить

```
10 DIM AQ 17
```

В этом случае получили бы

### КОРЕНЬ УРАВНЕНИЯ=2.5

В бейсике можно использовать не только текстовые переменные, но и текстовые массивы.

Введение текстовых массивов вполне естественная вещь: последовательность фамилий учеников, записанных в виде текста в памяти ЭВМ, можно считать одномерным текстовым массивом. Иногда удобно рассматривать таблицы, в графах которых содержатся тексты. Например, таблицу из двух граф, где в строках одной графы записаны фамилии, в строках другой — адреса проживания соответствующих лиц.

Такую таблицу можно считать матрицей текстовых переменных, состоящей из многих строк и двух столбцов. Описываются текстовые массивы точно так же, как и числовые. Например, запись

`DIM A$(3), B$(3,4)`

означает, что в программе будет использован массив с именем А, состоящий из трех текстовых переменных, и массив с именем В, представляющий матрицу из трех строк и четырех столбцов, элементами которой являются текстовые переменные. Все эти переменные могут иметь длину, не превышающую 16. Если мы хотим указать максимальную длину элементов массива, то делается это, как и прежде, заданием этого значения в операторе DIM. Например, если элементы массива А могут иметь максимальную длину 30, то мы должны написать

`DIM A$(3) 30`

В бейсике допускается сравнение не только числовых, но и текстовых величин. Довольно естественно считать, что две текстовые величины равны, если попарно совпадают все входящие в них символы, в том числе и пробелы. Так, если сравнить текстовые величины "ПАРОХОД" и "ПАРОВОЗ", то они не равны, так как у них символы, начиная с 5-го, не совпадают.

Выполним последовательность операторов

`10 DIM A$(17)`

`20 LET A$ = "КВАДРАТНЫЙ КОРЕНЬ"`

`30 LET B$ = "КВАДРАТНЫЙ КОРЕНЬ"`

`40 PRINT A$`

`50 PRINT B$`

На экране появится

КВАДРАТНЫЙ КОРЕНЬ

КВАДРАТНЫЙ КОРЕНЬ

Слово "КОРЕНЬ" во втором случае не высветилось полностью, так как максимальная длина текстовой переменной не оговорена и, следовательно, равна шестнадцати (с учетом пробелов). Иначе говоря, значения A\$ и B\$ не совпали.

Но, оказывается, можно сравнивать текстовые величины не только на совпадение, но и говорить о том, какая величина больше, какая меньше. Действительно, так как каждый символ в памяти ЭВМ представляется набором из восьми двоичных цифр, то этот набор можно рассматривать как двоичное число. Это позволяет утверждать, что одна буква больше другой (один символ больше другого), имея в виду, что соответствующие им величины, выраженные двоичным числом, одна больше другой. А коль скоро установлен порядок по величине между символами, можно говорить и о том, что одно слово больше другого, даже если число букв в них одинаково.

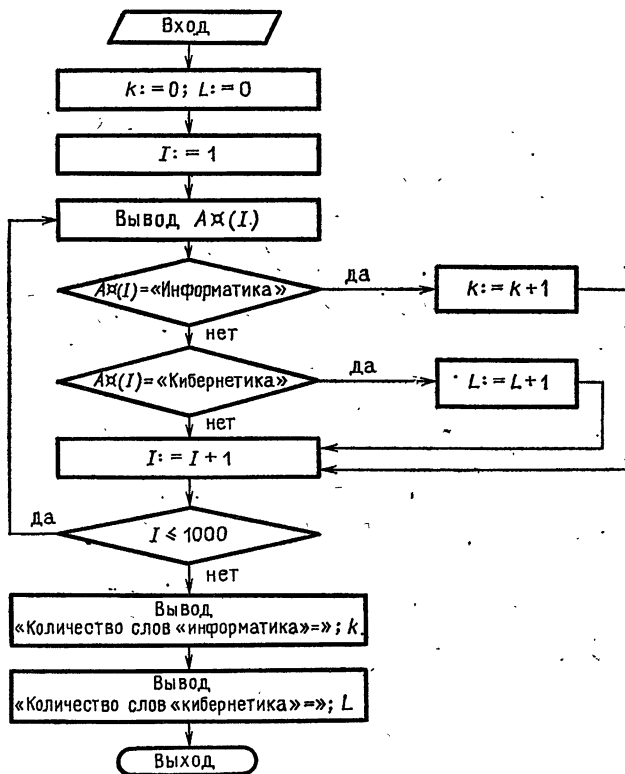


Рис. 6.1

Появление в языке бейсик операторов, сравнивающих слова на больше, меньше или равно, поэтому вполне правомочно. В дальнейших рассуждениях мы будем считать, что упорядоченность букв по алфавиту соответствует упорядоченности букв по величине. Это предположение на самом деле не всегда соответствует действительности —

в реальных ЭВМ величина кода буквы не обязательно соответствует величине ее номера в алфавитном порядке русского языка.

Итак, упорядочив все алфавитно-цифровые символы в порядке возрастания значений их кодов, мы можем считать, что у каждого символа есть свой ранг. И про любые два символа известно, у которого из них ранг больше, а у кого меньше. Пользуясь этим, можно сравнивать текстовые значения. Для этого сопоставляют крайние слева символы сравниваемых величин. Если один из них имеет больший ранг, то соответствующая текстовая величина считается большей. Так, например, можно сказать, что справедливо соотношение

НАДЯ < ТАНЯ

так как ранг символа Т больше ранга символа Н.

Если же первые символы имеют одинаковый ранг, то сравнивают между собой вторые символы и т. д. Например, верно соотношение

ВАНЯ < ВАСЯ

так как первые два символа здесь попарно совпадают, но ранг символа Н меньше ранга символа С.

Теперь мы уже можем решить такую простую задачу, как подсчитать количество вхождений некоторых слов в заданный текст. Пусть, например, в памяти ЭВМ находится некоторый текст из тысячи слов, и нам нужно подсчитать, сколько раз в этом тексте содержится слово «информатика» и сколько раз слово «кибернетика».

Составим блок-схему алгоритма (рис. 6.1).

Программа на бейсике выглядит следующим образом:

60 REM ОПРЕДЕЛИТЬ КОЛИЧЕСТВО СЛОВ

70 REM "ИНФОРМАТИКА" и "КИБЕРНЕТИКА"

80 LET K = 0

90 LET L = 0

100 REM K, L — счетчики слов

110 FOR I = 1 TO 1000

120 PRINT A\$(I)

130 IF A\$(I) = "ИНФОРМАТИКА" THEN 160

140 IF A\$(I) = "КИБЕРНЕТИКА" THEN 180

150 GOTO 190

160 LET K = K + 1

170 GOTO 190

180 LET L = L + 1

190 NEXT I

200 REM КОНЕЦ ПОДСЧЕТА СЛОВ

210 PRINT "КОЛИЧЕСТВО СЛОВ "ИНФОРМАТИКА" = ; K

220 PRINT "КОЛИЧЕСТВО СЛОВ "КИБЕРНЕТИКА" = ; L

С помощью сравнения текстовых величин можно решать задачу сортировки. Пусть нам, например, нужно выдать на печать два слова, расположив их в алфавитном порядке.

Составим программу на бейсике:

```
10 INPUT A□  
20 INPUT B□  
30 IF A□ > B□ THEN 70  
40 PRINT A□  
50 PRINT B□  
60 GOTO 90  
70 PRINT B□  
80 PRINT A□  
90 PRINT "КОНЕЦ СОРТИРОВКИ"
```

В строках 10 и 20 вводятся текстовые переменные A□ и B□. Затем оператор условного перехода IF осуществляет переход к строке 70 в случае, если значение A□ оказалось больше значения B□. Тогда печатается сначала значение B□ и в следующей строке экрана значение A□. Если же окажется, что значение A□ не больше значения B□, то условие A□ > B□ окажется несоблюденным и выполнятся операторы в строках 40 и 50, которые выдадут на экран сначала значение A□ и в следующей строке значение B□.

В любом случае программа завершится выдачей текста  
КОНЕЦ СОРТИРОВКИ

Выполним, например, эту программу для значений текстовых переменных A□ и B□ — ИВАНОВ и ПЕТРОВ.

Как мы знаем, пуск программы осуществляется набором на клавиатуре инструкции RUN. Оператор ввода в 10-й строке потребует информацию. На экране высветится знак вопроса:?. Наберем фамилию  
ИВАНОВ

Машина введет это значение, и оператор ввода в 20-й строке тоже потребует информацию — на экране опять появится знак вопроса:?. Наберем на клавиатуре

ПЕТРОВ

После этого выполнится оператор IF в 30-й строке. Условие A□ > B□ окажется несоблюденным. Поэтому вслед за оператором IF будут работать операторы в последующих строках, т. е. в 40-й, 50-й, 60-й и 90-й. В результате получим на экране

ИВАНОВ  
ПЕТРОВ  
КОНЕЦ СОРТИРОВКИ

Легко убедиться в том, что тот же результат получится, если введем фамилии в обратном порядке:

RUN  
? ПЕТРОВ  
? ИВАНОВ



Действительно, на этот раз условие  $A \square > B \square$  окажется выполненным и оператор IF осуществит переход на 70-ю строку, где произойдет вывод значения  $B \square$  (а в этой роли теперь выступит слово ИВАНОВ), а затем  $A \square$  (теперь это ПЕТРОВ).

Подобным образом мы теперь можем упорядочить в алфавитном порядке любой список. Пусть, например, надо расположить в алфавитном порядке фамилии десяти учеников: Лаптев, Иванов, Антонов, Петрова, Попова, Гуляев, Макаров, Борисов, Шундров, Федоров.

Обсудим сначала идею алгоритма. Прежде всего нам надо ввести в машину заданную последовательность фамилий. Это у нас будет

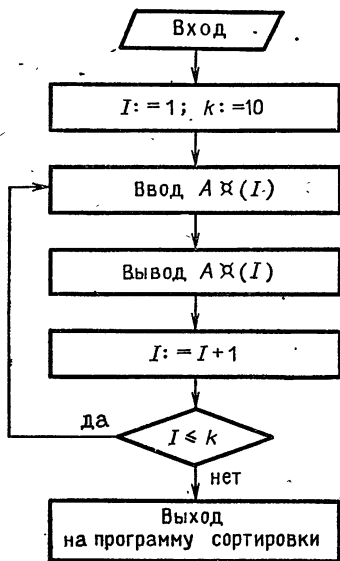


Рис. 6.2

первая часть программы. Затем нам надо упорядочить эти фамилии, т. е. выполнить собственно сортировку.

Что касается первой части, то, поскольку фамилий у нас теперь много, мы ввод организуем в виде циклического процесса последовательной записи фамилий в текстовой массив A.

В качестве переменного индекса будем использовать идентификатор I.

Блок-схема первого этапа программы представлена на рис. 6.2.

Как видно из блок-схемы, мы здесь не только вводим, но и выводим информацию на экран, чтобы убедиться в правильности ее ввода.

Теперь мы перейдем к рассмотрению второй части — к алгоритму сортировки. Будем исходить из той же идеи, что уже рассматривалась, а именно будем строить текстовый массив с упорядоченным расположением фамилий, выбираемых из исходного текстового массива  $A \square$ . То есть мы должны в массиве  $A \square$  так переставить элементы, чтобы они расположились в порядке возрастания их ранга.

Для решения этой задачи поступим следующим образом. Выберем из массива  $A \square$  элемент с наименьшим рангом и поменяем его местами с элементом, стоящим на первом месте.

Теперь у нас на первом месте стоит элемент  $A \square(1)$  с наименьшим рангом, и осталось упорядочить остальные элементы (их стало девять). Задача по существу свелась к предыдущей, т. е. надо выбрать элемент с наименьшим рангом, но уже среди девяти элементов. Найдя этот элемент, мы поменяем его местами с элементом  $A \square(2)$ .

Затем будем упорядочивать оставшиеся 8 элементов получившегося массива  $AQ$ . И так далее, до тех пор, пока все элементы массива  $AQ$  не окажутся расположенными в алфавитном порядке.

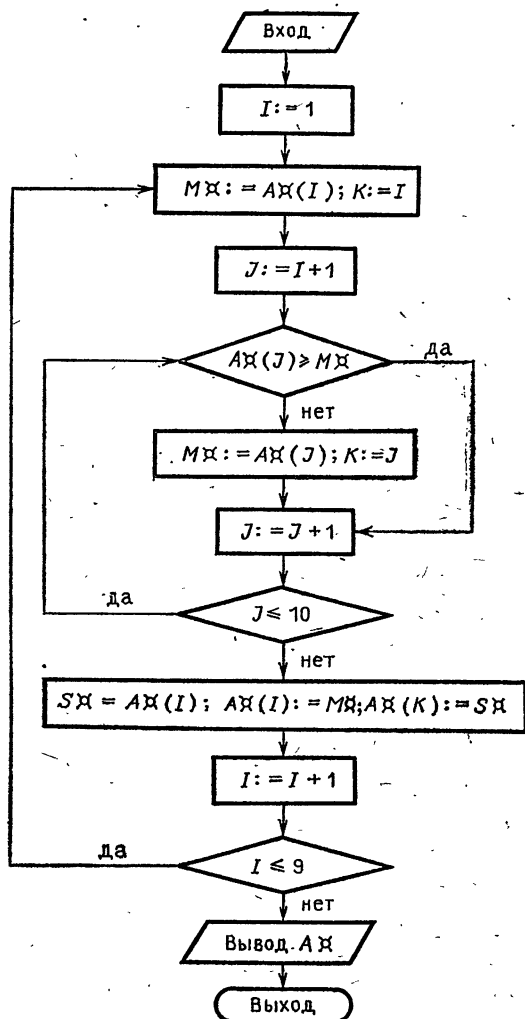


Рис. 6.3

Блок-схема этого участка алгоритма сортировки представлена на рис. 6.3.

В предложенном алгоритме есть два цикла — внутренний и внешний. Во внутреннем цикле происходит выбор элемента с наименьшим

рангом среди еще не упорядоченных элементов исходного массива. И этот элемент ставится среди них на первое место. А внешний цикл управляет тем, чтобы таким образом были упорядочены все элементы исходного массива  $A_{ij}$  — от первого до последнего.

Теперь этот алгоритм запишем на языке бейсик:

```
10 REM УПОРЯДОЧЕНИЕ СЛОВ ПО АЛФАВИТУ
20 DIM Aij (10)
30 FOR I=1 TO 10
40 INPUT Aij (I)
50 PRINT Aij (I)
60 NEXT I
70 REM ЗАКОНЧЕН ВВОД
80 REM НАЧАЛО УПОРЯДОЧЕНИЯ
90 FOR I = 1 TO 9
100 LET Mij = Aij (I)
110 LET K = I
120 REM НАЧАЛО ВНУТРЕННЕГО ЦИКЛА
130 FOR J = I + 1 TO 10
140 IF Aij (J) ≥ Mij THEN 170
150 LET Mij = Aij (J)
160 LET K = J
170 NEXT J
180 REM КОНЕЦ ВНУТРЕННЕГО ЦИКЛА
190 LET Sij = Aij (I)
200 LET Aij (I) = Mij
210 LET Aij (K) = Sij
220 NEXT I
230 REM КОНЕЦ ВНЕШНЕГО ЦИКЛА
240 REM ВЫВОД РЕЗУЛЬТАТА
250 PRINT
260 PRINT "*****"
270 PRINT
280 PRINT "СПИСОК ФАМИЛИЙ ПО АЛФАВИТУ"
290 FOR I = 1 TO 10
300 PRINT Aij (I)
310 NEXT I
320 STOP
```

Мы не будем приводить никаких комментариев к предложенной программе, так как подробные объяснения к блок-схеме алгоритма полностью применимы к программе на бейсике. Единственное замечание сделаем относительно операторов с номерами 250—280. Они нужны исключительно для того, чтобы отделить список фамилий от списка, полученного в результате работы программы, и, кроме того, озаглавить итоговый список.

Если мы теперь применим нашу программу к списку фамилий, упоминавшемуся ранее, то результатом работы программы будет

ЛАПТЕВ  
ИВАНОВ  
АНТОНОВ  
ПЕТРОВА  
ПОПОВА  
ГУЛЯЕВ  
МАКАРОВ  
БОРИСОВ  
ШУНДРОВ  
ФЕДОРОВ

\* \* \* \* \*

СПИСОК ФАМИЛИЙ ПО АЛФАВИТУ

АНТОНОВ  
БОРИСОВ  
ГУЛЯЕВ  
ИВАНОВ  
ЛАПТЕВ  
МАКАРОВ  
ПЕТРОВА  
ПОПОВА  
ФЕДОРОВ  
ШУНДРОВ

Заметим, что рассмотренный алгоритм можно применить и для решения задачи об упорядочении по возрастанию последовательности чисел. Для этого достаточно заменить в записи программы текстовые массивы на массивы чисел.

При работе над текстами часто бывает необходимо исследовать слово не целиком, а отдельные его символы или последовательность символов. Так бывает, если нам нужно выделить в тексте все слова, обладающие некоторым определенным признаком, например, начинающиеся на определенную букву, кончающиеся некоторым заданным буквосочетанием, имеющие один и тот же корень и т. д.

Длина каждой текстовой переменной, как мы знаем, задается с помощью оператора DIM, или же если мы не описываем эту переменную с помощью оператора DIM, то эта длина равна 16 символам. Если же нам надо использовать только часть символов, составляющих эту переменную, то для этого используется специальная текстовая функция STR (от string — строка), выделяющая из значения текстовой переменной последовательность символов. Эту последовательность часто называют «подстрокой», в отличие от текстовой переменной, которая задает «строку» символов.

Прежде чем дать общее определение функции STR, приведем конкретный пример ее использования. Так, если нам нужно в стро-

ке, являющейся значением переменной AQ, выделить подстроку, начинающуюся с 3-го символа и содержащую 5 символов, т. е. третий, четвертый, пятый, шестой и седьмой символы, то достаточно написать STR (AQ, 3, 5). Например, если значением AQ было ИНФОРМАТИКА, то в результате работы оператора

```
PRINT STR (AQ, 3, 5)
```

на экране дисплея появится слово

ФОРМА

Если же к текстовой переменной AQ применить оператор

```
PRINT STR (AQ, 3, 6)
```

то получим на экране дисплея

ФОРМАТ

Дадим теперь общее определение функции STR. Для обращения к функции STR следует написать

STR (<текстовая переменная>, <арифм. выражение 1>, <арифм. выражение 2>)

либо

STR (<текстовая переменная>, <арифм. выражение 1>)

Текстовая переменная здесь может, в частности, быть элементом текстового массива.

<Арифметическое выражение 1> задает начальный номер символа выделяемой подстроки. <Арифметическое выражение 2> определяет количество символов в выделенной подстроке. Значения обоих арифметических выражений обязаны быть целыми.

Приведенные выше примеры поясняют это определение функции STR. Но, как видно из определения, <арифметическое выражение 2> может отсутствовать. Тогда выделяемая подстрока состоит из всех оставшихся справа символов текстовой переменной, начиная с символа, задаваемого арифметическим выражением 1. Так, если применить к нашей текстовой переменной AQ оператор

```
PRINT STR(AQ,6)
```

то получим набор символов

МАТИКА

Разумеется, значения арифметических выражений должны быть не меньше единицы, значение выражения 1 — не больше количества символов в соответствующем текстовом значении, а значение выражения 2 — не больше количества оставшихся справа символов в текстовом значении.

Функция STR может применяться во всех операторах языка, где используются текстовые переменные. В частности, как в левой,

так и в правой части оператора присваивания. Например, пусть значения переменных A□ и B□ определены следующим образом:

```
20 LET A□ = "ОЦЕНКА"  
30 LET B□ = "ОТЛИЧНО"
```

Тогда в результате выполнения оператора присваивания

```
40 LET STR(A□, 8) = B□
```

переменная A□ получит значение

ОЦЕНКА ОТЛИЧНО

Функцию STR можно использовать и для проверки условия в операторе IF. Например, в результате выполнения оператора

```
50 IF STR(A□, 4, 2) = "МА" THEN 120
```

осуществится переход к строке с номером 120, если в текстовой переменной A□ символ под номером 4 представляет собой М, а символ с номером 5 — соответственно А. В противном случае будет выполняться оператор из следующей строки.

Рассмотрим простой пример. Пусть нам в ходе решения некоторой задачи понадобилось выбрать из некоторого текстового массива S и выдать на экран дисплея все слова, начинающиеся с буквы К. Пусть также известно, что длина массива равна 500.

Для решения этой задачи достаточно использовать в программе следующие операторы:

```
.....  
150 FOR I=1 TO 500  
160 IF STR(S□(I), 1, 1) <> "K" THEN 180  
170 PRINT S□(I)  
180 NEXT(I)
```

(Знак < > означает «не равно».)

В приведенном выше фрагменте программы произойдет последовательный просмотр всех элементов текстового массива S□. Если очередной элемент не начинается с буквы К, то происходит просто переход к просмотру следующего элемента. Если же условие в операторе IF не соблюдено, т. е. очередной элемент массива S□ начинается с К, то выполнится оператор вывода в 170-й строке и этот элемент появится на экране.

Рассмотрим задачу упорядочения слов в алфавитном порядке по первой букве. Пусть, как и раньше, у нас есть некоторая последовательность фамилий, которую мы хотим расположить в алфавитном порядке. Будем считать, что фамилии могут начинаться с любой буквы русского алфавита, кроме Ъ и Ь, и фамилий в списке всего 20.

Тогда для решения поставленной задачи возможен такой вариант программы на бейсике:

```
10 REM УПОРЯДОЧИТЬ ПО ПЕРВЫМ БУКВАМ АЛФАВИТА
20 DIM A$(20), B$(20), L$31
30 LET L$ = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХ
  ЦЧШЩЫЭЮЯ"
40 FOR I=1 TO 20
50 INPUT A$(I)
60 PRINT A$(I)
70 NEXT I
80 REM ОКОНЧЕН ВВОД
85 LET K=1
90 FOR J=1 TO 31
100 FOR I=1 TO 20
110 IF STR(L$,J,1) <> STR(A$(I),1,1) THEN 140
120 LET B$(K)=A$(I)
130 LET K=K+1
140 NEXT I
150 REM КОНЕЦ ПРОВЕРКИ ПО ОДНОЙ БУКВЕ
160 NEXT J
170 REM КОНЕЦ УПОРЯДОЧЕНИЯ
174 PRINT
175 PRINT "СПИСОК ПО АЛФАВИТУ"
176 PRINT
180 FOR I=1 TO 20
190 PRINT B$(I)
200 NEXT I
210 REM КОНЕЦ. ВЫВОД РЕЗУЛЬТАТА
```

В программе операторами от 40-й до 70-й строки осуществляется ввод и вывод исходной информации. Затем в программе идут вложенные циклы. Внутренний составляют операторы с 100-й строки по 140-ю. Здесь начальные буквы всех слов сравниваются с одной очередной буквой алфавита. Так, при  $J=1$  такое сравнение происходит с буквой А (значением  $STR(L$,1,1)$  является А). Если встретится слово, начинающееся с А, то оно заносится в массив В\$. Счетчиком компонент массива В\$ служит переменная К. Как только в В\$ заносится очередной элемент, значение К увеличивается на единицу. После того как произойдет последовательное сравнение начальных букв всех слов с буквой А, завершится выполнение внутреннего цикла и во внешнем цикле параметр цикла J увеличится на единицу. Затем снова начинает работу внутренний цикл. Опять просматриваются все слова списка, но их первые буквы сравниваются теперь уже со второй буквой алфавита (значением  $STR(L$,2,1)$  является уже Б). Все слова из списка, начинающиеся с этой буквы, поочередно заносятся

в массив ВQ и т. д. Таким образом, перебираются все слова списка и все буквы алфавита. Вслед за этим результирующий массив ВQ, в котором к данному моменту все слова расположены в алфавитном порядке (напомним, что в расчет при упорядочении принимались только первые буквы слов), выводится на экран.

Мы можем несколько упростить программу, если не требовать образования текстового массива ВQ, а просто выбирать из массива АQ слова в алфавитном порядке и выводить их на экран дисплея:

```

10 REM.УПОРЯДОЧИТЬ ПО ПЕРВЫМ БУКВАМ АЛФАВИТА
20 DIM AQ(20),LQ31
30 LET LQ="А Б В Г Д Е Ё Ж З И Й К Л М Н О П Р С Т У Ф Х
  Ц Ч Ш Щ Ъ Э Ю Я"
40 FOR I=1 TO 20
50 INPUT AQ(I)
60 PRINT AQ(I)
70 NEXT I
80 REM ОКОНЧЕН ВВОД
90 PRINT
100 PRINT "СПИСОК ПО АЛФАВИТУ"
110 PRINT
120 FOR J=1 TO 31
130 FOR I=1 TO 20
140 IF STR(LQ,J,1) < STR(AQ(I),1,1) THEN 160
150 PRINT AQ(I)
160 NEXT I
170 NEXT J

```

Данная программа осуществляет вывод элемента текстового массива АQ(I) на экран сразу же, как только его первая буква совпадает с той, с которой в данный момент осуществляется сравнение.

Обе приведенные программы не претендуют на оптимальность. Действительно, если все двадцать фамилий списка начинаются, например, только с букв А и Б, то они высветятся все на экране сразу же после проверки на совпадение их начальных букв с буквами А и Б, т. е. задача будет решена. Однако даже после этого все равно будут последовательно сравниваться первые буквы фамилий списка со всеми остальными буквами алфавита.

Этого недостатка можно избежать, если, например, в программе предусмотреть прекращение упорядочения после того, как число упорядоченных слов совпадет с общим количеством слов в списке.

В последних примерах, работая с переменными текстового массива, мы точно знали, в каких позициях этих переменных могут находиться интересующие нас символы. Так, например, мы искали символы МА в 4-й и 5-й позициях. При отборе слов, начинающихся с К,



и при упорядочении фамилий по алфавиту нас интересовали символы, находящиеся в первой позиции обрабатываемых слов.

Однако далеко не всегда известны заранее номера позиций, где располагаются интересующие нас символы. Представим себе, что у нас есть массив текстовых переменных разной длины. Необходимо выбрать и вывести на экран те из них, которые оканчиваются последовательностью букв КА. При этом длина исследуемых слов может быть разной. Предположим только, что она не меньше двух и не превышает какого-либо фиксированного числа (например, слова могут содержать не более двадцати символов).

В этом случае заранее неизвестно, какими по счету являются последние два символа обрабатываемых слов. А ведь именно эти символы мы должны сравнивать с КА. Действительно, если в слове ИНФОРМАТИКА последние два символа стоят в 10-й и 11-й позициях, то в слове РУБКА — соответственно в 4-й и 5-й.

Мы знаем, что прежде, чем работать с текстовым массивом, необходимо с помощью оператора DIM задать максимальную длину входящих в него переменных. Тогда значение каждой переменной может содержать в своей записи и меньше символов, чем это максимально допустимо, а недостающие до максимального количества символы заполняются пробелами. Например, если мы описали текстовый массив с помощью оператора

`DIM A$(10)20`

а массив A\$ состоит из 10 элементов, длина каждого из которых не превышает 20, то переменная этого массива — "ИНФОРМАТИКА" имеет девять пробелов в конце, переменная "РУБКА" — пятнадцать таких пробелов, а переменная "ОЦЕНКА: ОТЛИЧНО" имеет пять концевых пробелов, т. е. пробелов, идущих подряд вслед за последним символом О в слове ОТЛИЧНО.

Как мы уже отмечали, нас интересуют те символы, которые непосредственно предшествуют концевым пробелам, иначе говоря, последние значащие символы. Для того чтобы их отыскать, нужно знать длину переменных без учета количества концевых пробелов. Для определения количества символов, записанных в текстовую переменную без учета концевых пробелов, используется функция LEN (length — длина). Ее значение равно числу символов в записи переменной без концевых пробелов. Аргументом функции LEN является либо сама текстовая переменная, либо значение функции STR от этой переменной.

Использовать функцию LEN может в различных операторах и выражениях. Например,

```
20 IF LEN(A$) > 10 THEN 100
50 LET STR(B$(1, LEN(C$))) = X$
90 PRINT LEN(STR(C$, 2))
```

В 20-й строке выполняется оператор IF, который осуществляет переход к 100-й строке программы, если количество значащих символов текстовой переменной AQ больше десяти.

В 50-й строке выполняется оператор присваивания, который первым компонентам переменной BQ присваивает первые символы значения переменной XQ. Количество присваиваемых байтов задается длиной хранимой в CQ последовательности символов без учета концевых пробелов. Наконец, в 90-й строке находится оператор PRINT, который выводит на экран число, равное количеству символов в записи переменной CQ без учета ее первого символа.

Теперь мы можем вернуться к нашей задаче о выборе из массива текстовых значений тех из них, которые оканчиваются символами КА. Итак, предположим, что массив с именем AQ состоит из десяти текстовых значений, максимальная длина которых не превышает двадцати. Выведем на экран те из них, которые оканчиваются на КА.

Запишем программу на бейсике:

```
10 REM СЛОВА, ОКАНЧИВАЮЩИЕСЯ НА "КА"
20 DIM AQ(10)20
30 FOR I=1 TO 10
40 INPUT AQ(I)
50 PRINT AQ(I)
60 NEXT I
65 PRINT
70 PRINT "СЛОВА, ОКАНЧИВАЮЩИЕСЯ НА "КА""
75 PRINT
80 FOR I=1 TO 10
90 LET B=LEN(AQ(I))
100 IF STR(AQ(I), B-1,2) <> "КА" THEN 120
110 PRINT AQ(I)
120 NEXT I
```

Операторы в строках 30—60 осуществляют ввод исходного массива в ЭВМ и вызов его элементов на экран дисплея. Операторы с 80-го по 110-й проверяют все переменные на совпадение двух последних их символов с КА. И те из переменных, которые оканчиваются на КА, оператором с номером 110 выводятся на экран. Так, для последовательности переменных "ЛИТЕРАТУРА", "ФИЗИКА", "МАТЕМАТИКА", "АСТРОНОМИЯ", "ГЕОГРАФИЯ", "ИСТОРИЯ", "ИНФОРМАТИКА", "ЛОГИКА", "ОБЩЕСТВОВЕДЕНИЕ", "АРИФМЕТИКА" получим

ЛИТЕРАТУРА  
ФИЗИКА  
МАТЕМАТИКА  
АСТРОНОМИЯ  
ГЕОГРАФИЯ

ИСТОРИЯ  
ИНФОРМАТИКА  
ЛОГИКА  
ОБЩЕСТВОВЕДЕНИЕ  
АРИФМЕТИКА

СЛОВА, ОКАНЧИВАЮЩИЕСЯ НА "КА"

ФИЗИКА  
МАТЕМАТИКА  
ИНФОРМАТИКА  
ЛОГИКА  
АРИФМЕТИКА

Эту задачу нам удалось решить с помощью функций STR и LEN. Оператор в 90-й строке программы присваивает переменной В значение длины текстовой переменной A(I), т. е. тем самым по существу определяется номер последнего символа в записи A(I). Нам же нужно было выделить два последних символа переменной A(I). Они, естественно, находятся на (В-1)-й и В-й позициях. Например, при I=2 имеем

A(I) = "ФИЗИКА"

и

LEN(A(I)) = 6

Поэтому В=6 и значение В-1 будет равно 5, т. е. в 100-й строке программы оператор IF проверяет совпадение с КА двух последних значащих символов переменной A(I) (в нашем случае 5-го и 6-го символов). Причем условие в операторе считается выполненным, если сравниваемые символы не совпадают. В этом случае происходит просто переход к проверке следующего элемента текстового массива. В нашем случае, т. е. при I=2, условие несовпадения символов не соблюдается. Поэтому выполняется оператор вывода текстовой переменной A(I), расположенный в 110-й строке, и лишь потом происходит переход к обработке следующего элемента. Аналогичным образом происходит исследование остальных элементов массива.

Как мы уже говорили, перечисленные выше средства работы с текстами позволяют не только редактировать текст, но и оформлять текстовый документ для последующего вывода на печатающее устройство так, чтобы он выглядел красиво, либо удовлетворял некоторым правилам расположения печатного материала на листе бумаги. В деловых документах часто требуется, чтобы тексты были размещены в рамках, определенных граф, чтобы справа (или слева) были оставлены поля определенного размера и т. д. Операторы работы с текстовыми переменными позволяют писать программы, которые будут автоматически приводить исходный текст к нужному формату.

Такой процесс называется форматированием текста, а программы, выполняющие такую работу, называют *форматерами*.

При создании программ-форматеров часто возникают задачи, связанные с теми или иными способами выравнивания текстов.

Рассмотрим пример построения программы, выравнивающей тексты по правому краю.

Пусть у нас имеется некоторый набор слов, каждое из которых надо вывести на экран так, чтобы их последние буквы были расположены строго одна под другой, т. е. нужно выравнивать слова по их правому краю. В качестве таких слов возьмем, например, следующие: ИНФОРМАТИКА, АЛГОРИТМ, БАЙТ, БИТ.

Составим следующую программу на бейсике:

```
10 REM ВЫРАВНИВАНИЕ ПО ПРАВому КРАЮ
20 DATA 4, "ИНФОРМАТИКА", "АЛГОРИТМ"
30 DATA "БАЙТ", "БИТ"
40 READ N
50 FOR I=1 TO N
60 READ A()
70 PRINT TAB(20-LEN(A()));STR(A(),1,LEN(A()))
80 NEXT I
```

Результатом работы программы будет

```
ИНФОРМАТИКА
  АЛГОРИТМ
    БАЙТ
      БИТ
```

Прокомментируем программу.

Операторы DATA формируют блок данных, в который войдет число 4 и текстовые величины "ИНФОРМАТИКА", "АЛГОРИТМ", "БАЙТ" и "БИТ". С помощью оператора READ в 40-й строке переменная N получает значение 4. Операторы, находящиеся в строках 60 и 70, выполняются в цикле. В теле цикла происходит чтение очередного текстового значения (оператор READ в 60-й строке), а затем вывод этой величины в нужном месте экрана. Для этого используется оператор PRINT с функциями TAB и LEN.

Напомним, что функция TAB задает номер позиции в строке экрана, начиная с которой будут выводиться символы, идущие вслед за функцией TAB в операторе PRINT. Например, в результате работы оператора

```
PRINT TAB(5); "ТАБЛИЦА"
```

на экране слово ТАБЛИЦА высветится, начиная с 5-й позиции строки экрана.

Функция LEN(A()) вычисляет длину значения текстовой переменной A(). Например, для слова "ИНФОРМАТИКА" значение

функции будет равно 11. В этом случае значением аргумента функции TAB будет число 9:

$$20 - \text{LEN}(\text{A}\square) = 20 - 11 = 9.$$

Поэтому слово "ИНФОРМАТИКА" выводится с 9-й позиции строки экрана. Аналогично, слово "АЛГОРИТМ" имеет длину 8. Следовательно, это слово выводится, начиная с 12-й позиции экрана ( $20 - 8 = 12$ ). Слово "БАЙТ" выводится, начиная с 16-й позиции ( $20 - 4 = 16$ ), а "БИТ" — начиная с 17-й ( $20 - 3 = 17$ ). Тем самым последние символы всех выводимых слов окажутся в одной и той же позиции экрана — в 19-й, но в разных строках.

Для многих приложений важна задача подсчета частоты встречаемости букв в тексте. Например, расположение букв на клавиатуре пишущей машинки выбрано из соображений частоты встречаемости букв и их сочетаний. В середине клавиатуры сосредоточены наиболее часто встречающиеся буквы. Такое расположение экономит затраты энергии человека, работающего за клавиатурой, и повышает производительность труда.

В связи с этим рассмотрим следующую задачу. Пусть имеется некоторый текст из четырех предложений, каждое из которых представляется последовательностью не более чем 50 символов. Задача состоит в том, чтобы сосчитать количество гласных в этом тексте. Будем считать, что у нас есть массив  $\text{A}\square$ , состоящий из 4 текстовых значений.

Составим программу на бейсике:

```
10 REM КОЛИЧЕСТВО ГЛАСНЫХ В ТЕКСТЕ
20 DIM A□(4)50
30 FOR I=1 TO 4
40 INPUT A□(I)
50 PRINT A□(I)
60 NEXT I
70 LET G□="АЕЁИОУЫЭЮЯ"
80 LET N=0
90 FOR I=1 TO 4
100 LET Q=LEN(A□(I))
110 FOR J=1 TO Q
120 FOR K=1 TO 10
130 IF STR(A□(I),J,1) < > STR(G□,K,1) THEN 150
140 LET N=N+1
150 NEXT K
160 REM КОНЕЦ ПЕРЕБОРА ГЛАСНЫХ
170 NEXT J
180 REM КОНЕЦ ПЕРЕБОРА БУКВ ПЕРЕМЕННОЙ
190 NEXT I
195 PRINT "*****"
200 PRINT "КОЛИЧЕСТВО ГЛАСНЫХ=";N
```

В программе используются вложенные циклы. Самый внешний цикл с параметром цикла  $I$  нужен для перебора предложений. В нашей задаче таких предложений четыре. Так, при  $I=1$  просматривается первое предложение, при  $I=2$  — второе предложение и т. д. Внутри этого цикла в свою очередь вложенные циклы. Цикл с параметром  $J$  служит для просмотра всех символов внутри одного предложения. Поэтому  $J$  пробегает значения от единицы до  $Q = \text{LEN}(A(I))$  — номера последнего символа в текстовой переменной  $A(I)$ , т. е. в  $I$ -м предложении. При каждом значении  $J$  выполняется еще один цикл — цикл с параметром  $K$ , в котором символ, стоящий в  $J$ -й позиции исследуемого предложения, последовательно сравнивается со всеми десятью гласными буквами. И как только этот символ совпадает с одной из них, к счетчику гласных букв добавится единица.

Применим нашу программу к следующему тексту:  
НАША СЕМЬЯ СОСТОИТ ИЗ ШЕСТИ ЧЕЛОВЕК:  
ОТЕЦ РАБОТАЕТ ТРАКТОРИСТОМ,  
МАТЬ — ДОЯРКА НА МОЛОЧНОЙ ФЕРМЕ,  
ДЕТИ УЧАТСЯ В ШКОЛЕ

Так, в нашем случае сначала  $I=1$  и будет просмотрено первое предложение. Переменная  $J$  получит значение единица. Это означает, что будет исследоваться первый символ первого предложения. У нас это буква  $H$ , которая будет теперь последовательно сравниваться со всеми гласными буквами, набор и порядок следования которых определены оператором с номером 70. Очевидно, что ни с одной из гласных букв  $H$  не совпадет. Поэтому параметр  $J$  увеличится на единицу, т. е. теперь в том же первом предложении (так как  $I$  все еще равно единице) будет исследоваться второй символ. В нашем случае это буква  $A$ , которая будет последовательно сравниваться со всеми гласными буквами. При первом же таком сравнении — с буквой  $A$  произойдет совпадение символов и значение счетчика гласных букв — переменной  $N$  увеличится на единицу. До сих пор он был равен нулю, теперь станет равным единице. Ясно, что ни с какой другой буквой исследуемый нами символ уже сравнивать далее нет смысла — он с ними заведомо не совпадает. Вот тут как раз видно несовершенство нашей программы. Она будет продолжать сравнение  $J$ -го символа, т. е. буквы  $A$  с остальными гласными буквами из зафиксированного нами списка, пока не переберет их все. И лишь только после этого перейдет к следующему символу в предложении, т. е. значение переменной  $J$  увеличится на единицу. Теперь буква  $Ш$  будет сравниваться со всеми гласными буквами и т. д. Процесс будет происходить до тех пор, пока мы не просмотрим все символы, составляющие первое предложение. После этого переменная  $I$  увеличится на единицу и начнется просмотр второго предложения. Так будет до тех пор, пока не завершится просмотр всего текста.

Результат работы программы на экране дисплея будет выглядеть так:

НАША СЕМЬЯ СОСТОИТ ИЗ ШЕСТИ ЧЕЛОВЕК:  
ОТЕЦ РАБОТАЕТ ТРАКТОРИСТОМ,  
МАТЬ—ДОЯРКА НА МОЛОЧНОЙ ФЕРМЕ,  
ДЕТИ УЧАТСЯ В ШКОЛЕ

\*\*\*\*\*  
КОЛИЧЕСТВО ГЛАСНЫХ=40

Итак, мы убедились, что даже тот небольшой набор операторов и функций, который мы здесь рассмотрели, представляет довольно широкие возможности по работе с текстовой информацией.

## ГЛАВА 7

### МАШИННАЯ ГРАФИКА

До сих пор мы рассматривали способы вывода на экран дисплея лишь такой информации, которая представлялась в алфавитно-цифровом виде. Это были числа, таблицы, тексты и даже графики функций в виде набора отдельных точек этих графиков, представленных каким-либо символом. Однако при решении многих задач как исходными данными, так и результатами являются схемы, чертежи, рисунки, графики и т. д. Поэтому весьма естественно иметь возможность отображать эти изображения (графики) на экране. Такая возможность имеется во многих ЭВМ. Современные персональные компьютеры снабжены цветными графическими экранами, на которых можно не только высвечивать алфавитно-цифровую текстовую информацию, но и выводить цветные рисунки, графики и даже создавать мультипликации.

В каких случаях оказывается необходимым использовать ЭВМ для построения изображений? В наше время резко возрос объем проектно-конструкторских работ. Проектируются новые сооружения, новые станки и механизмы, самолеты и космические корабли. Проекты включают в свой состав не только описания изделий, но и огромное количество схем и чертежей. Тысячи чертежников заняты сложной и кропотливой работой. Если подсчитать затраты на чертежное хозяйство по всей стране, то они окажутся очень большими, да и труд чертежника, хотя он и требует навыка и умения, нельзя считать в полной мере творческим и интересным. Если к этому прибавить также и то, что даже высококвалифицированный специалист с большим опытом затрачивает очень много времени на эту работу, то станет ясным — этап создания чертежей задерживает общее время проектирования.

ЭВМ, снабженная устройствами вывода графической информации и соответствующим комплексом программ, такую работу может выполнить намного быстрее и более качественно. Конструктор, используя графические возможности ЭВМ, может вывести на экран дисплея чертеж некоторого узла механизма и с помощью специальных программ



рассчитать, например, его прочностные характеристики. В случае нужды конструктор тут же у дисплея внесет в чертеж необходимые изменения, после чего может снова провести соответствующие расчеты. Зачастую машина может представить чертеж изучаемого механизма в нужном конструктору ракурсе, т. е. как бы поворачивать исследуемый узел механизма на экране. ЭВМ может предложить конструктору на выбор несколько решений, подсказать ему оптимальное решение той или иной задачи проектирования.

Несколько слов о принципах работы дисплея. Его устройство весьма напоминает экран цветного телевизора. Как вы знаете, такой экран состоит из множества люминофорных точек, которые начинают светиться в тот момент, когда на них попадает электронный луч. Эти точки могут светиться синим, зеленым и красным цветом. Точки так близко расположены друг от друга, что с некоторого расстояния складывается впечатление непрерывной картинки. В этом же смысле можно говорить и о построении «непрерывных графиков» и «гладких картинок» на цветном экране персональных ЭВМ.

Электронно-лучевая трубка дисплея покрыта с внутренней стороны люминофором — светящимся веществом. Так называемая электронная пушка создает узкий лучок электронов — электронный луч. Подавая на электронную пушку ток разного напряжения, можно увеличивать или уменьшать интенсивность электронного луча. Попадая на люминофор экрана в некоторую его точку, луч сильной интенсивности возбуждает соответственно сильное свечение этой точки, слабый луч возбуждает слабое свечение. Электронный луч вообще можно на какое-то время «выключить», и свечения не будет. У основания луча по выходе его из электронной пушки в электронно-лучевой трубке устанавливаются устройства, способные отклонять луч «вправо, влево, вверх, вниз». Обычно это электромагниты или пластины, создающие электростатическое напряжение. Подавая на отклоняющую систему ток той или иной характеристики, можно заставить луч описывать на экране различные кривые. Если заставить ЭВМ подавать на электронную пушку и на отклоняющую систему электронно-лучевой трубки в определенной последовательности соответствующие напряжения, то можно заставить луч рисовать на экране вполне осмысленные картинки.

Мы знаем, что ЭВМ обрабатывает данные, представленные в двоичном виде, т. е. в виде наборов нулей и единиц. Эти наборы обозначают числа, буквы, различного рода признаки. В виде двоичных данных могут быть представлены также и закодированные изображения. Специальные логические электронные схемы, выбирая из памяти ЭВМ коды изображений, преобразуют их в сигналы, управляющие движением электронного луча по экрану, регулирующие его интенсивность. Таким образом, чтобы получить нужные изображения на экране, необходимо составить алгоритмы и программы, которые со-

здадут в памяти закодированный рисунок. Высвечиванием рисунка на экране занимается электроника, связанная с электронно-лучевой трубкой и получающая необходимую информацию из памяти ЭВМ.

В различных ЭВМ способы кодирования рисунков в памяти и техника их высвечивания на экране заметно отличаются. Например, существуют так называемые векторные дисплеи, в которых луч наподобие карандаша вычерчивает линии на экране. Но наибольшее распространение получили растровые дисплеи, построенные по типу телевизоров. Луч в этих устройствах обходит экран регулярным образом, строка за строкой, с одинаковой скоростью. Чтобы нарисовать рисунок, в этом случае следует управлять не движением луча (оно всегда одно и то же), а его интенсивностью. Своевременно увеличивая интенсивность, можно получить на одной строке сильно светящиеся и слабо светящиеся участки. Эти участки, сливаясь на экране по всем его строкам, образуют видимую картинку. Следует при этом сказать, что хотя в растровых экранах луч в своем движении последовательно по времени обходит экран строка за строкой, но это происходит столь быстро, что складывается впечатление устойчивой картинки. Люминофор в растровых экранах нанесен обычно в виде близко лежащих друг от друга точек на строке. Картинка тем самым складывается из точек разной степени яркости и цвета.

Современные дисплеи могут работать в двух режимах: алфавитно-цифровом и графическом.

Расскажем более подробно о создании графического изображения на экране ЭВМ.

На каждом типе экрана может быть высвечено определенное количество точек, с помощью которых и строится рисунок. Например, экран ЭВМ «Искра-226» имеет 256 строк, каждая из которых содержит по 512 точек, т. е. всего  $256 \times 512 = 2^{17}$  точек. Дисплей персональной ЭВМ «Ямаха» имеет 192 строки по 256 точек в строке.

Для того чтобы указать любую точку на экране, достаточно задать номер строки, в которой она находится, и порядковый номер ее в этой строке, т. е. по существу надо указать координаты этой точки. Системы координат в разных ЭВМ используются разные. Например, в компьютере «Ямаха» начало координат помещается в левом верхнем углу экрана, ось  $x$  направлена вправо, ось  $y$  — вниз (рис. 7.1), а в ЭВМ «Искра-226» положение начала координат зафиксировано в левом нижнем углу, ось  $x$  направлена вправо, ось  $y$  — вверх (рис. 7.2).

Как уже отмечалось, для того чтобы получить на экране какое-либо изображение, необходимо заставить светиться соответствующие точки экрана.

Можно считать, что луч, высвечивающий точки на экране, может перемещаться в любую другую точку, двигаясь по экрану от одной

соседней точки к другой, т. е. совершая некоторый путь «мелкими» шагами. При этом на своем пути он может высвечивать точки своего пути, а может их и не высвечивать, если в момент прохождения через точку его интенсивность падает до нуля.

Расстояние между соседними точками называется *дискретом*. При этом дискрет по строке может отличаться по величине от дискрета по вертикали. Как правило, стараются сделать экран таким образом, чтобы эти дискреты по вертикали и горизонтали почти не отличались друг от друга. Для обычных дисплеев этот дискрет равен приблизительно 0.4 мм:

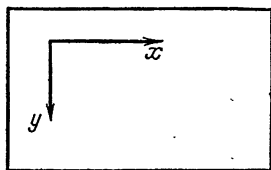


Рис. 7.1

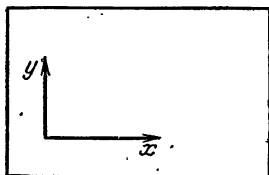


Рис. 7.2

Мы уже говорили о том, что для того чтобы заставить электронику дисплея рисовать нужные нам картинки на экране, надо закодировать желаемое изображение и запомнить этот код в памяти ЭВМ.

Как же кодируется изображение? Самый простой способ — это задать в памяти последовательность координат тех точек, по которым должен двигаться электронный луч, при этом не забывая каждый раз указывать — высвечивать данную точку на его пути или не высвечивать. Это в свою очередь означает, что мы должны уметь составить программу, которая будет заполнять память такой информацией. Дело это кропотливое и сложное. Для того чтобы упростить процесс кодирования изображений, в языки программирования вводятся специальные команды, позволяющие оперировать не только с отдельными точками, а с так называемыми графическими объектами. Подобно тому, как легче строить дом из крупных блоков, чем из отдельных кирпичиков, так и изображение на экране проще строить не точка за точкой, а из более крупных «конструкций» — отрезков прямых линий, частей окружностей, квадратов. Сопрягая между собой эти конструкции, можно получить более сложные изображения. Для того чтобы задать, например, отрезок прямой, надо указать координаты его начала и конца — это проще, чем задать координаты всех точек на экране, через которые пройдет эта прямая. Для того чтобы нарисовать часть окружности, достаточно задать координаты центра, координаты точки начала дуги и значение угла сектора.

Конструкции, из которых строятся все другие изображения, называются *элементарными графическими объектами*. Наборы элементарных графических объектов, включаемых в операторы языка, в разных системах отличаются, но чаще всего используются та-

кие: точка, отрезок прямой, дуга окружности (или окружность целиком), прямоугольник. Из указанных выше элементарных графических объектов можно строить рисунки букв, цифр и любых других символов, а также более сложные графические объекты. В частности, можно, например, выводить тексты символов из графических объектов произвольного размера с любыми расстояниями между ними. «Графический объект» — это очень широкое понятие. К нему можно отнести часть рисунка на экране, весь кадр, высвечиваемый на экране, характерную фигуру, отдельную букву или целый текст.

Некоторые графические объекты заранее заготовлены. Например, к ним можно отнести рисунки букв и символов стандартного текста на экране. Если речь идет об обычном рисунке на бумаге, например, о фрагменте чертежа какой-либо детали, то чертежник может перерисовать эту деталь в увеличенном масштабе, может на новом листе объединить в единое целое чертежи нескольких деталей, может их чертить в разных ракурсах, различным образом повернутыми друг относительно друга. Естественно поэтому создать и в машине средства, которые позволяли бы выполнять на экране аналогичные действия с графическими объектами, т. е. производить с ними те же манипуляции, которые может выполнять профессионал-чертежник. А это значит, что в наборе команд ЭВМ должны быть такие команды, при помощи которых она может выполнять действия по рисованию на экране. Такие действия включаются в язык бейсик в виде специальных операторов машинной графики. С их помощью можно из элементарных графических объектов строить произвольные графические объекты.

Сделаем здесь одно важное замечание. Мы уже говорили ранее, что используемые в разных ЭВМ версии языка бейсик могут отличаться друг от друга. Проявляется это и в операторах машинной графики, т. е. на разных машинах для построения одних и тех же графических объектов могут использоваться различные наборы операторов. Мы дадим здесь одну из версий расширения бейсика в область машинной графики. При работе на любой конкретной ЭВМ, прежде чем пользоваться средствами машинной графики, необходимо тщательно изучить соответствующую инструкцию (описание языка) и пользоваться теми операторами, которые допустимы в данной версии языка.

Во многих системах простейшим и в то же время наиболее универсальным графическим оператором является PLOT (график). С его помощью можно изобразить на экране дисплея произвольную конфигурацию точек, т. е. нарисовать все, что угодно. Координаты этих точек задаются в качестве параметров оператора PLOT. С точки зрения программиста в этом операторе задается последовательность точек на экране, отражающая путь воображаемого карандаша. Оператор PLOT как бы указывает следующую точку, в которую надо передвигать

нуть этот воображаемый карандаш, относительно того места на экране, на котором он в данный момент находится.

Существует понятие «графического курсора», движение которого по экрану как раз и задается оператором PLOT. Эти движения графического курсора могут быть заданы самым замысловатым образом: можно вырисовывать сложные фигуры, буквы различных размеров и стилей написания, плотно заштриховывать целые области на экране. В качестве параметров оператора PLOT задается число дискрет по горизонтали и число дискрет по вертикали, на которые следует передвинуть графический курсор, с тем чтобы он попал в следующую точку, предназначенную для высвечивания. Вы знаете, что реально на экране дисплея точки высвечиваются благодаря тому, что в них направляется фокусированный электронный луч электронно-лучевой трубки. С некоторым приближением можно сказать, что оператор PLOT управляет движением электронного луча от одной точки экрана к другой. «С некоторым приближением» означает то, что в действительности процесс высвечивания точек на экране более сложен и в разных типах экранов он происходит по-разному. Но принципиально (логически) этот процесс можно представлять, как было сказано выше.

Рассмотрим теперь подробнее, как записывается оператор PLOT в программе и какие он выполняет действия в зависимости от вида его записи.

Для того чтобы перевести воображаемый рисующий карандаш — графический курсор — в начало координат, следует написать оператор

PLOT ( , , R)

В результате выполнения этого оператора графический курсор переводится в начало координат дисплея, например, на ЭВМ «Искра-226» — это левый нижний угол экрана.

Курсор может перемещаться по экрану из одной точки в другую, не оставляя следа на экране, но можно дать задание прочертить, т. е. высветить, путь от одной точки до другой в виде отрезка прямой.

Если мы хотим переместить курсор из какой-то точки экрана в другую, которая отличается от текущей по оси  $x$  на  $\Delta X$  шагов (дискрет), а по оси  $y$  на  $\Delta Y$  шагов, то мы должны написать либо

PLOT ( $\Delta X$ ,  $\Delta Y$ , U)

если курсор перемещается, не оставляя следа, либо

PLOT ( $\Delta X$ ,  $\Delta Y$ , D)

если курсор оставляет светящийся след.

Пусть, например, нам надо начертить прямоугольный треугольник, катеты которого направлены параллельно осям координат и составляют соответственно 50 дискрет по оси  $x$  и 100 дискрет по оси  $y$ .

С помощью оператора PLOT это можно выполнить следующим образом:

```
10 PLOT (.,R)
20 PLOT (200,100,U)
30 PLOT (0,100,D)
40 PLOT (50,—100,D)
50 PLOT (—50,0,D)
```

Результат работы программы представлен на рис. 7.3. Точка слева внизу рисунка показывает начало координат.

Прокомментируем программу. Оператор в 10-й строке переводит графический курсор в начало координат. Оператор в 20-й строке переводит его на 200 дискрет вправо по оси  $x$  и на 100 дискрет вверх по оси  $y$ , т. е. определяет одну из вершин треугольника. Оператор в 30-й строке строит катет по вертикали. Затем оператор в 40-й строке рисует гипотенузу треугольника, смещая графический курсор на 50 дискрет вправо и на 100 дискрет вниз. Наконец, последний оператор строит второй катет треугольника. Подчеркнем, что каждый оператор указывает смещение относительно текущей точки.

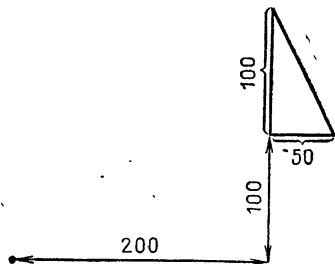


Рис. 7.3

На самом деле не было необходимости выписывать пять операторов PLOT. Бейсик допускает запись этой последовательности в виде одного оператора с указанием списка параметров. В нашем случае это будет

```
10 PLOT (.,R),(200,100,U),(0,100,D),(50,—100,D),(—50,0,D)
```

Очень часто при рисовании графиков и чертежей требуется кроме самого рисунка графика размещать на экране различного рода надписи, заголовки, цифры. Каждый раз вырисовывать оператором PLOT все необходимые знаки довольно затруднительно. Чтобы упростить такого рода работу, в средства машинной графики включается специальная модификация оператора PLOT, позволяющая более простым образом задавать «вырисовывание» текстов.

Покажем, как с помощью такого рода модификации оператора можно рисовать на экране различные тексты.

Для того чтобы изобразить на экране какую-либо строку символов, достаточно написать

```
PLOT ( $\Delta X, \Delta Y$ , «строка символов»)
```

В этом случае строка символов отобразится на экране, начиная с позиции, отличающейся от текущей точки на  $\Delta X$  дискрет по горизонтали и  $\Delta Y$  дискрет по вертикали. Причем стандартные размеры

каждого символа — 5 дискрет по горизонтали и 7 — по вертикали. Так, если мы хотим вывести строку символов ТЕКСТ, разместив ее с 300-й позиции в 150-й строке экрана, то мы должны задать следующий оператор:

```
10 PLOT (, ,R), (300,150,"ТЕКСТ")
```

На экране высветится слово  
ТЕКСТ

Этот же результат мы могли бы, правда, получить, используя и оператор PRINT с функцией TAB. Но если нам понадобится вывести текст, где размеры символов не были бы стандартными, то оператор PRINT нам уже не поможет. Как же можно задавать размеры символов в бейсике?

Масштаб символов устанавливается с помощью оператора вида  
PLOT(*n*, ,C)

Здесь C — параметр, говорящий о том, что данный оператор PLOT используется именно для установления масштаба изображения символов. Число *n* задает масштаб. Если параметр *n* не указан в операторе, то масштаб равен единице.

Таким образом, после выполнения оператора

```
10 PLOT (, ,R), (2, ,C), (300, 150, "ТЕКСТ")
```

на экране получим слово

ТЕКСТ

Таким образом, размеры символов увеличились вдвое по сравнению с предыдущим примером.

Рассмотрим теперь следующую программу:

```
10 REM ПРОГРАММА ПИШЕТ ТЕКСТ
20 FOR I=1 TO 10
30 PLOT (, ,R), (15, ,C), (30+I,50+2*I,"МИР")
40 NEXT I
50 STOP
```

Результат работы этой программы приведен на рис. 7.4. Как видно из рисунка, каждая буква нашего текста состоит из 10 ломаных линий. Каждая из этих линий рисуется отдельно при соответствующем значении параметра цикла I. Например, при I=1 оператор PLOT устанавливает курсор в начало координат, назначает масштаб символов  $n=15$  и, смещая точку на 31 дискрет по оси *x* и 52 дискрета по оси *y*, рисует слово МИР в заданном масштабе. При следующем значении параметра цикла, т. е. при I=2, курсор опять устанавливается в начало координат, снова берется масштаб  $n=15$  и опять пишется слово МИР, только теперь с 32-й позиции по горизонтали и с 54-й по вертикали. И так происходит 10 раз, причем

каждое следующее написание слова МИР происходит со смещением на один дискрет по горизонтали и два дискрета по вертикали.

Однако оператор PLOT не всегда удобен для практического использования. Во-первых, рисование фигур точка за точкой—вещь обременительная и требует большой предварительной проработки.

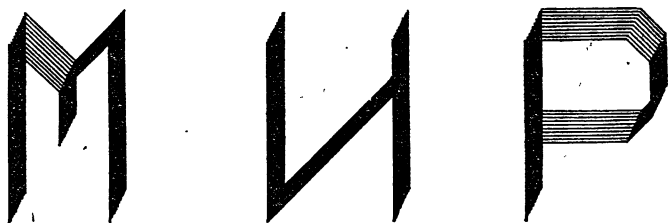


Рис. 7.4

По существу, сначала все, что вы хотите нарисовать, надо начертить вручную на бумаге в клеточку, чтобы точно определить начала и концы экранных координат. Во-вторых, с помощью оператора PLOT мы не могли задать абсолютные координаты точки на экране, а лишь указывали смещение на  $\Delta X$  и  $\Delta Y$  дискрет относительно текущей точки.

Для того чтобы расширить возможности машинной графики, в бейсике используют целый ряд специальных графических операторов. Они дополняют оператор PLOT возможностью создавать графические объекты из графических элементов (точка, вектор, текст), производить над этими объектами различные преобразования—сжатие, растяжение, сдвиг, поворот.

Сложные рисунки на экране можно создавать из отдельных, заранее заготовленных графических объектов или, что то же самое, из отдельных кусков изображений.

Представим себе, что перед нами стоит задача рисовать на экране различные шахматные позиции. Для этого нам надо заготовить изображения шахматных фигур, черных и белых, изображение шахматной доски. Если мы сумели заранее все это заготовить, то дело изображения конкретной позиции будет сведено к размещению уже заранее созданных графических объектов в нужном порядке. В данном случае наши графические объекты—изображения фигур.

Заготовленные графические объекты размещаются в памяти ЭВМ в виде массивов данных (символьных массивов) и в процессе компоновки позиции извлекаются из памяти специальными операторами.

Заготавливаемым впрок графическим объектам даются собственные имена (идентификаторы). Это позволяет в дальнейшем обращаться к графическому объекту по присвоенному ему имени. Таким образом, для того чтобы использовать в программе какой-либо графический объект, необходимо его предварительно объявить (дать ему имя). Это



делается с помощью оператора OPEN (открыть). Общий вид этого оператора:

OPEN SQ0

Данный оператор говорит о том, что в программе будет использован графический объект с именем SQ.

Для соответствующего символьного массива, предназначенного для хранения графического объекта, предварительно необходимо зарезервировать память с помощью оператора DIM. Если оператор DIM не писать, то под соответствующий символьный массив для графического объекта отведется 1600 байт.

Рассмотрим теперь простой пример. Пусть нам надо построить графический объект, представляющий собой треугольник с вершинами в точках (100, 50), (170, 110) и (150, 180). Эту задачу можно решить и с помощью оператора PLOT. Но тогда нам придется предварительно рассчитать, на сколько позиций по горизонтали и вертикали надо смещаться при переходе от одной вершины к другой. Гораздо удобнее было бы, если бы мы имели возможность просто указать абсолютные координаты той точки на экране, в которую мы должны попасть из текущей точки.

Такую возможность предоставляет оператор DRAW (чертить). Его общий вид:

DRAW SQ0, X, Y

Здесь SQ — имя формируемого графического объекта, (X, Y) — абсолютные координаты конечной точки вектора, который проводится из текущей точки формируемого объекта.

Если, например, выполнить следующую последовательность операторов:

```
10 OPEN SQ0
20 PLOT (, R)
30 DRAW SQ0 (, 100, 50
```

то на экране высветится вектор, идущий из начала координат в точку с координатами (100, 50) (рис. 7.5).

Если же нам нужно попасть в какую-либо точку экрана, ничего не рисуя, то мы можем сделать это с помощью оператора NPLLOT. Его общий вид:

NPLLOT SQ0 (, X, Y

Выполнив этот оператор, мы попадем в точку с координатами (X, Y). Поэтому, если нам надо провести вектор из точки (100, 50) в точку (170, 110), мы должны выполнить следующую последовательность операторов:

```
10 OPEN SQ0
20 NPLLOT SQ0, 100, 50
30 DRAW SQ0, 170, 110
```

Первый оператор здесь объявит, что в программе будет использован графический объект SQ, второй оператор переводит графический курсор (не оставляя на экране никакого следа) в точку с координатами (100, 50), и, наконец, третий оператор рисует вектор из текущей точки, т. е. точки (100, 50), в точку с координатами (170, 110).

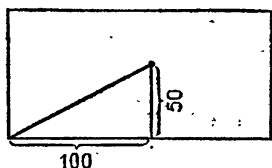


Рис. 7.5

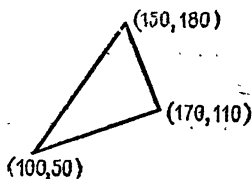


Рис. 7.6

Теперь мы можем решить задачу о построении треугольника. Программа на бейсике выглядит так:

```
10 REM ПОСТРОЕНИЕ ТРЕУГОЛЬНИКА
20 OPEN SQ0,
30 REM ОТКРЫТЬ GO
40 NPL0T SQ0, 100, 50
50 REM ВЫХОД В НАЧАЛЬНУЮ ТОЧКУ
60 DRAW SQ0, 170, 110
70 DRAW SQ0, 150, 180
80 DRAW SQ0, 100, 50
90 REM ТРЕУГОЛЬНИК НАРИСОВАН
100 STOP
```

Результат работы программы показан на рис. 7.6.

Таким образом, с помощью этой программы мы создали графический объект, который назвали SQ. Он представляет собой треугольник с заданными сторонами, расположенный вполне определенным образом на экране. Если мы захотим использовать этот графический объект, в качестве элемента более сложного рисунка, мы должны иметь возможность указать, куда этот объект переместить на экране, как его повернуть, вообще, как его преобразовать.

В бейсике есть возможность преобразовывать графические объекты. Для этого используются следующие операторы: MOVE (движение, смещение), TURN (поворот) и STRETCH (растяжение).

Общий вид оператора MOVE:

```
QMOVE SQ0, ΔX, ΔY
```

Здесь SQ — имя сдвигаемого графического объекта, ΔX — величина

сдвига в дискретах по оси  $x$ ,  $\Delta Y$ —величина сдвига в дискретах по оси  $y$ .

Например, если в предыдущей программе после строки 80 поставить оператор

```
85  $\square$ MOVE  $\square$ 0, 200, -20
```

то после его выполнения построенный ранее треугольник сдвинется на экране на 20 дискрет вправо по оси  $x$  и на 20 дискрет вниз по оси  $y$ . Если же нужно повернуть графический объект относительно какой-либо точки, то используют оператор TURN, общий вид которого следующий:

```
TURN  $\square$ 0, X, Y,  $\phi$ 
```

Здесь  $\square$ —имя графического объекта, который нужно повернуть,  $(X, Y)$ —координаты точки экрана, относительно которой осуществляется поворот,  $\phi$ —угол в радианах, на который поворачивается графический объект. При этом положительное значение угла соответствует вращению против часовой стрелки.

Так, если продолжать преобразования нашего треугольника, то, чтобы повернуть его на  $90^\circ$ , достаточно выполнить оператор

```
86 TURN  $\square$ 0, 300, 30, 1.57
```

Точка с координатами (300, 30) задает ту вершину треугольника, которая получилась после перемещения вершины (100, 50) на  $\Delta X = 200$  и  $\Delta Y = -20$  в результате выполнения оператора MOVE. Именно вокруг этой вершины осуществляется поворот на  $90^\circ$ —(в радианах — это  $\pi/2$ ).

Наконец, если нужно растянуть или сжать графический объект, то используют оператор STRETCH. Его общий вид:

```
STRETCH  $\square$ 0, X, Y,  $k_x$ ,  $k_y$ 
```

Здесь  $\square$ —имя графического объекта,  $(X, Y)$ —координаты точки на экране, относительно которой производится растяжение (или сжатие),  $k_x$ —коэффициент растяжения (сжатия) вдоль оси  $x$ ,  $k_y$ —коэффициент растяжения (сжатия) вдоль оси  $y$ .

Продолжим преобразовывать наш графический объект—треугольник. В результате выполнения оператора

```
87 STRETCH  $\square$ 0, 300, 30, .5, .5
```

треугольник сожмется вдвое относительно вершины (300, 30).

Приведем теперь в целом программу построения и преобразования треугольника:

```
10 REM ПОСТРОЕНИЕ ТРЕУГОЛЬНИКА
```

```
20  $\square$ OPEN  $\square$ 0
```

```
30 NPLT  $\square$ 0, 100, 50
```

```
40 DRAW  $\square$ 0, 170, 110
```

```

50 DRAW SQ0, 150, 180
60 DRAW SQ0, 100, 50
70 REM ТРЕУГОЛЬНИК ПОСТРОЕН
80 MOVE SQ0, 200, -20
90 REM ТРЕУГОЛЬНИК СДВИНУТ
100 TURN SQ0, 300, 30, 1.57
110 REM ТРЕУГОЛЬНИК ПОВЕРНУТ
120 STRETCH SQ0, 300, 30, .5, .5
130 REM ТРЕУГОЛЬНИК СЖАТ
140 STOP

```

На рис. 7.7 представлены результаты работы программы.

Заметим, что во всех наших примерах координаты (X, Y) точек в операторах, смещения ( $\Delta X$ ,  $\Delta Y$ ) задавались в виде целых чисел — констант. В действительности их можно задавать и с помощью переменных или даже арифметических выражений.

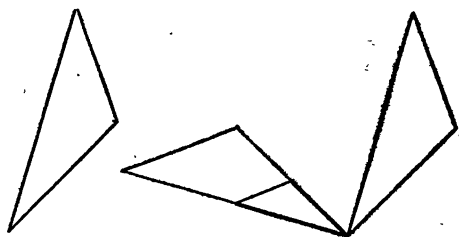


Рис. 7.7

Например, если выполнить последовательность операторов

```

10 OPEN SQ0
20 LET X0=250
30 LET Y0=120
40 LET R=80
50 NPLOT SQ0, X0+R, Y0

```

то графический курсор попадет в точку экрана с координатами ( $X0+R$ ,  $Y0$ ), т. е. в точку (330, 120).

Попробуем теперь решить еще одну задачу — нарисовать окружность радиуса  $R=80$  с центром в точке (250, 120).

Эту задачу можно решить с помощью следующей программы:

```

10 REM ПОСТРОЕНИЕ ОКРУЖНОСТИ
20 OPEN SQ0
30 LET X0=250
40 LET Y0=120
50 LET R=80
60 NPLOT SQ0, X0+R, Y0

```

```

70 REM ВЫШЛИ В НАЧАЛЬНУЮ ТОЧКУ
80 FOR F=0 TO 6.2832 STEP 0.0349
90 DRAW SC0, X0+R*COS(F), Y0+R*SIN(F)
100 REM РИСОВАНИЕ ОКРУЖНОСТИ
110 NEXT F
120 STOP

```

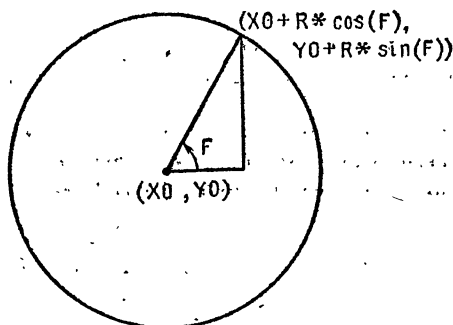


Рис. 7.8

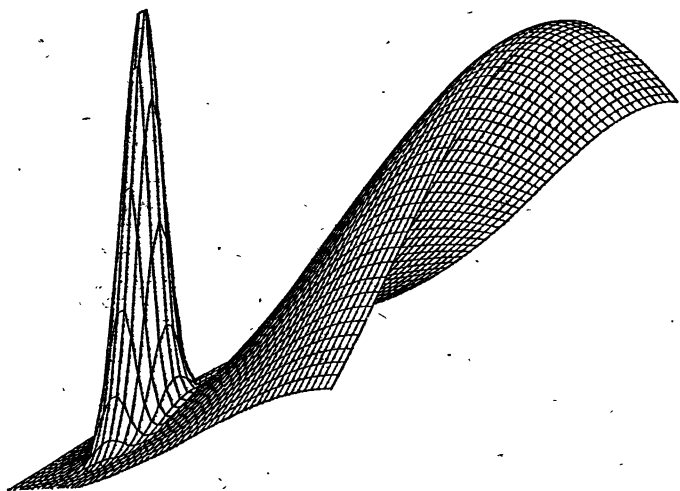


Рис. 7.9

Окружность (рис. 7.8) здесь рисуется циклическим выполнением оператора DRAW в 90-й строке программы.

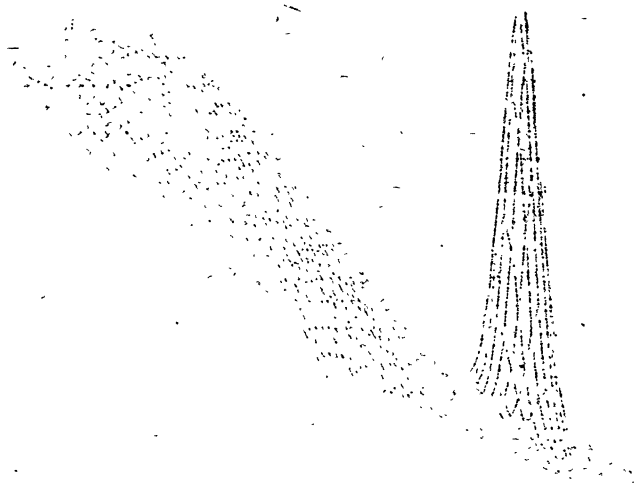
В некоторых версиях бейсика есть более удобные средства построения окружности. Например, в машине «Ямаха» для того, чтобы нарисовать окружность, достаточно выполнить оператор

**CIRCLE(X, Y, R)**

где  $X$  и  $Y$  задают координаты центра, а  $R$  — радиус окружности.

Мы здесь изложили лишь незначительную часть тех возможностей для машинной графики, которые существуют в различных версиях языка бейсик. В частности, мы совершенно не затронули цветной графики, рассмотрели мало операторов для построения графических объектов.

Чтобы читатель мог убедиться в богатых возможностях машинной графики, мы приведем рисунок (рис. 7.9), полученный с использованием графических операторов на ЭВМ «Искра-226».



## ГЛАВА 8

# ВЫЧИСЛИТЕЛЬНЫЕ МАШИНЫ И ИХ МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ

В гл. 3 мы рассмотрели некоторую гипотетическую вычислительную машину с упрощенной системой команд, сравнительно легко понимаемым «машинным» языком. Эта машина, как мы уже говорили, похожа на ЭВМ первого поколения М-20. Мир вычислительных машин, а точнее вычислительных систем, чрезвычайно разнообразен. Разнообразен по функциональному назначению различных типов ЭВМ, по внутренней их организации (структуре), по машинному языку, по составу базового программного и математического обеспечения. К семейству вычислительных машин с программным управлением относятся и простейшие микрокалькуляторы и супер-ЭВМ, быстродействие которых превышает миллиард высокоточных операций в одну секунду. В этот мир входят вычислительные системы, содержащие в своем составе от двух-трех микропроцессоров до нескольких тысяч процессорных элементов, способные решать задачи, к которым два десятилетия назад нельзя было и подступиться. Как это обычно бывает, в случае большого разнообразия объектов, их стараются расклассифицировать, разделить на какие-то группы. Деление производят по разным критериям. Классификация ЭВМ необходима по той причине, что это помогает более объективно сравнивать типы ЭВМ между собой и, следовательно, обоснованно выбирать для применения в конкретных условиях ту или иную машину.

Мы уже знакомы с классификацией ЭВМ по поколениям. Эта классификация носит, так сказать, исторический характер, и в ее основе лежит элементная база, на которой ЭВМ строилась.

Машины последнего десятилетия нашего века будут принадлежать к ЭВМ V поколения, построенным на схемах сверхбольшой интеграции. Об этом поколении следует сказать несколько слов.

Технология производства интегральных схем в настоящее время достигла такого совершенства, что позволяет на одном кремниевом кристалле площадью в несколько десятков квадратных миллиметров размещать около миллиона элементарных электрических переключе-

ющих схем — вентилей. Это означает, что на один такой кристалл можно возложить логические функции, которые ранее выполнялись весьма крупными вычислительными машинами. Существенно то, что это дает возможность реально объединить много таких мощных логических узлов в единую вычислительную систему, содержащую сотни и тысячи «высокоорганизованных» логических модулей. Такую вычислительную систему можно заставить выполнять логические построения очень большой сложности, т. е. заставить делать многоступенчатые логические выводы и заключения, основанные либо на правилах формальной логики, либо на заранее заданных шаблонах. Последнее обстоятельство очень важно. Шаблоны «машинных» рассуждений могут задаваться на основе жизненного и профессионального опыта специалистов: научных работников, врачей, высококвалифицированных рабочих, сотрудников управлений различного ранга. Это означает, что в ЭВМ можно будет вложить человеческие знания и опыт.

Машины V поколения должны стать машинами логического вывода — рассуждающими машинами. Это не фантастика, уже сегодня на машинах III и IV поколения реализованы так называемые экспертные системы, вобравшие в себя профессиональный опыт нескольких сфер творческой производственной деятельности. Особое развитие получили диагностические экспертные системы: в медицине, в области практической экономики, учитывающей человеческий фактор, и даже в направлении прогнозирования наличия полезных ископаемых по таким косвенным проявлениям, как состав флоры и фауны.

Ведутся работы, которые позволят ЭВМ не только синтезировать речь (это делается уже сейчас), но и понимать вопросы, заданные на естественном языке. Таковы основные усилия ученых и инженеров, создающих ЭВМ новых поколений.

Другой разрез классификации ЭВМ связан с их внутренней организацией. Прежде всего, вычислительные машины подразделяются на однопроцессорные и многопроцессорные.

Для однопроцессорных ЭВМ характерно то, что они любую программу выполняют последовательно, операция за операцией. Предыдущее высказывание не совсем точно. В классе однопроцессорных ЭВМ существуют такие ЭВМ, выполнение операций в которых совмещено во времени, однако и в них никакая следующая по программе операция не может обогнать предыдущую. В Советском Союзе первой однопроцессорной ЭВМ с глубоким совмещением операций явилась замечательная машина БЭСМ-6, созданная по проекту С. А. Лебедева еще в 1967 г. Глубина совмещения операций в этой ЭВМ достигала 7, т. е. в некоторые периоды времени машина обрабатывает одновременно до 7 машинных команд, строго следя при этом за тем, чтобы не была нарушена логика их последовательного выполнения. Образно такой тип параллельно-последовательной обработки



можно представить рис. 8.1. Здесь горизонтальными отрезками представлены времена выполнения соответственно 1, 2, ..., 7 команд на ЭВМ.

Внутренняя организация глубокого совмещения операций позволила в 2—3 раза поднять производительность БЭСМ-6 по сравнению с другими ЭВМ, построенными на той же элементной базе. Система команд машины БЭСМ-6—одноадресная. Одноадресность является одним из способов экономии числа обращений к оперативной памяти при выполнении команд.

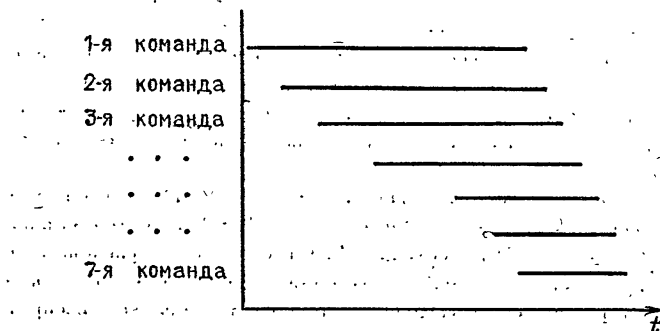


Рис. 8.1

В машине типа М-20 для модификации адресов использовался один регистр адреса, распространявший, если в этом была необходимость, свое влияние на все три адреса. В БЭСМ-6 таких регистров насчитывается 16. Это дает дополнительную гибкость при построении циклических программ. Номер используемого в команде регистра указывается в специальном четырехразрядном поле команды. Таким образом, хотя система команд называется одноадресной, в каждой инструкции указывается два адреса: адрес операнда и номер регистра. Код адреса операнда служит для формирования полного логического адреса путем прибавления к этому коду содержимого соответствующего регистра. Правильно поэтому называть код адреса операнда смещением относительно той адресной базы, которая записана в регистре. И в самом деле, код адреса операнда в БЭСМ-6 занимает всего 12 двоичных разрядов, а полностью сформированный адрес 15 двоичных разрядов. Такая система адресации со смещением принята во многих машинах. В частности, в системе команд ЕС ЭВМ. Код смещения в ЕС ЭВМ занимает 12 разрядов, а полный адрес 24 двоичных разряда. В БЭСМ-6 адресуется каждое машинное слово, состоящее из 6 байтов. В ЕС ЭВМ адресуется каждый байт, т. е. принята более детальная адресация.

Читатель даже по таким неполным данным о системе команд некоторых современных машин уже может составить представление о том,

как усложнился машинный язык по сравнению с машинным языком ЭВМ первого поколения. Эти усложнения естественно отразились и на структуре автокодов, в связи с чем программирование на машинно-ориентированных языках требует очень высокой квалификации. К сожалению, пока еще приходится пользоваться этими языками уровня тонкой детализации при создании так называемого системного программного обеспечения.

Вернемся, однако, к обзору различных средств классификации ЭВМ. Многопроцессорные вычислительные системы в свою очередь подразделяются на несколько типов в зависимости от того, каким образом связаны между собой их процессорные элементы. Самый простой и понятный способ такой связи — это объединение автономно работающих ЭВМ линиями передачи данных. Это позволяет ЭВМ обмениваться блоками информации в условиях, когда каждая ЭВМ выполняет свою собственную задачу по своей собственной программе.

Вычислительные системы, образованные из ЭВМ, связанных по каналам связи, чаще всего называют сетями ЭВМ, реже многомашинными комплексами. ЭВМ, объединенные в сети, могут располагаться друг от друга на значительном расстоянии, в разных городах и даже в разных странах. Такие сети называются территориально распределенными. В сеть могут увязываться несколько ЭВМ, расположенных в одном здании, в одном вычислительном центре. Такие сети называются локальными сетями.

Увязка ЭВМ в сети позволяет более эффективно использовать совокупную вычислительную мощность, разумно распределяя между ними весь тот объем вычислений, который подлежит выполнить в течение какого-то заданного промежутка времени (суток, недели, месяца). Другая особенность использования сетей, состоящих из машин разного типа и производительности, состоит в том, что если в каком-то городе или районе нет нужной по производительности ЭВМ, задание на выполнение работы можно передать по линиям связи в другой город, где необходимая ЭВМ имеется, и получить результаты по тем же линиям связи. Другими словами, сети позволяют организовать дистанционный доступ к ЭВМ.

Следующий тип вычислительных систем, состоящих из нескольких процессоров, — это векторные и матричные процессоры.

Предположим, что надо покомпонентно сложить два вектора:  $a_1, a_2, \dots, a_n$  и  $b_1, b_2, \dots, b_n$ . На однопроцессорной машине приходится эту процедуру выполнять последовательно. Сначала сложить  $a_1$  и  $b_1$ , затем  $a_2$  и  $b_2$  и т. д. Но если бы в нашем распоряжении была ЭВМ с  $n$  арифметико-логическими устройствами (АЛУ) и с каждым АЛУ была бы связана собственная память, содержащая компоненты  $a_i$  и  $b_i$ , то можно было бы всем АЛУ дать одну общую команду — сложить соответствующие компоненты векторов. На такой

многопроцессорной машине результат получился бы в  $n$  раз быстрее. Такие машины реально созданы, но они естественно приспособлены для решения задач, основное содержание которых составляют векторные вычисления. Отличительной чертой таких специализированных машин является то, что все их процессорные элементы (в нашем случае АЛУ) выполняют в каждый момент времени одинаковую для всех команду, но с разными данными, подобно тому как строй солдат выполняет одну команду «на-пле-чо», но каждый солдат при этом поднимает на плечо свою собственную винтовку.

Бывает и другой тип связей процессорных элементов, когда каждый из них работает по своей собственной программе, но все они имеют возможность обращаться к общей памяти. Здесь труднее придумать образную аналогию. Представьте себе группу ювелиров, каждый из которых делает свое особое украшение, а жемчужины все они выбирают из общего ларя. Если же к этому добавить, что все эти разные изделия предназначены для украшения одного костюма, то легко представить сложность взаимоотношений ювелиров, возникающих при организации их слаженной работы. Может быть, этот пример не столь уж и удачный, но в мультипроцессорных системах, предназначенных для решений одной общей задачи, расчлененной на параллельно исполняемые участки, возникают именно сложности в организации слаженной работы в синхронизации и регулировании моментов обращения к общей памяти. Такого рода вычислительные системы реально существуют. К ним, например, относится советская вычислительная система Эльбрус, в состав которой могут входить до 10 мощных процессоров, каждый производительностью около 12 млн. операций в секунду.

Вычислительные системы, состоящие из многих процессорных элементов: ЭВМ, АЛУ, центральных процессоров, работающих параллельно, создаются для того, чтобы в конечном итоге получить повышение производительности системы в целом, т. е. чтобы быстрее решать задачи. Необходимость повышения производительности диктуется темпами развития научно-технического прогресса. Например, чтобы своевременно обрабатывать данные, получаемые со спутников, необходимы вычислительные мощности в миллиарды операций в секунду. На одном процессоре, сделанном даже на сверхвысокочастотных элементах, достичь такого быстродействия весьма сложно. Остается один путь — распараллелить решение задачи. Способы распараллеливания для разных задач могут сильно отличаться. Отсюда возникает большое количество различных архитектур вычислительных систем, предназначенных для решения вполне определенного круга задач. Реально получается так, что разработчики аппаратуры ЭВМ сейчас создают ЭВМ, «подгоняя» их устройство (структуру и архитектуру) под несколько типов различных алгоритмов, под существующие математические методы решения задач. Можно вполне определенно сказать, что современные

ЭВМ есть детище тесного взаимодействия математической науки и вычислительной инженерии.

Теперь вкратце рассмотрим, что понимается под математическим обеспечением ЭВМ. Под *математическим обеспечением* (МО) понимается комплекс программ, обеспечивающих интерфейс (связь) с пользователем и управление аппаратными средствами вычислительной системы.

Интерфейс пользователя с ЭВМ служит нескольким целям, главная из которых состоит в упрощении способов общения человека с ЭВМ; в обеспечении простоты программирования, отладки задач, простоты доступа к ЭВМ.

Программные средства управления аппаратурой служат главной цели — оптимизации выбранных критериев эффективности использования вычислительной системы в различных режимах ее эксплуатации. Соответственно этому математическое обеспечение вычислительных систем принято делить на две части: *системы программирования* и *операционные системы* (ОС). Иногда, впрочем, эти две части объединяют общим названием ОС. Вообще говоря, в состав математического обеспечения включаются все программы, доступные для использования любому пользователю или в том или ином смысле обеспечивающие работу программ любого пользователя. Иными словами, все «общедоступные» программы многократного использования.

Состав и содержание математического обеспечения ЭВМ во многом определяется режимом использования ЭВМ. Различаются следующие режимы:

- однопользовательский, однозадачный режим использования;
- пакетный однопрограммный и мультипрограммный режим;
- режим мультидоступа;
- режим реального масштаба времени.

Первый, однопользовательский режим применяется на микро- и мини-ЭВМ, на персональных компьютерах различного назначения: бытовых, учебных, профессиональных. Характерной особенностью такого стиля использования ЭВМ является прямое взаимодействие пользователя с ЭВМ, монопольное владение всей вычислительной установкой в сеансе работы за клавиатурой пульта управления. Пользователь ведет диалог с ЭВМ, задавая ей вопросы, получая ответы и в свою очередь отвечая на вопросы и замечания, исходящие от ЭВМ. Математическое обеспечение в этом случае, рассчитанное на непосредственное взаимодействие с пользователем, строится таким образом, чтобы максимально упростить его работу. Вопросы, которые ЭВМ задает своему единственному хозяину, обычно сводятся к предложению выбрать для работы тот или иной язык, ту или иную служебную программу, входящую в состав программного обеспечения. ЭВМ предлагает пользователю меню — список тех услуг, которые она может выполнить. Пользователь выбирает из этого списка то, что ему нужно в данный момент, задает необходимые параметры и приступает к работе.

Персональные ЭВМ, как мы знаем, все снабжены телевизионным экраном, на котором высвечиваются тексты диалога, результаты работы. Экран и клавиатура в этом случае выполняют роль основного средства взаимодействия. Важную часть математического обеспечения в этом случае играют системные программы, обеспечивающие удобную работу с экраном, и группа программ, «понимающая» указания пользователя.

Список допустимых команд обычно достаточно велик и достигает в некоторых системах сотен. В этих директивах предусмотрены возможности задавать компьютеру указания на выполнение целого ряда работ, которые необходимы при формировании программ, их отладке, модификации и исполнении.

Следующий режим, называемый пакетным, характеризуется тем, что пользователь лишается прямого общения с ЭВМ. Он, сидя за обычным столом, составляет нужную ему программу, отдает ее на перфорацию (пробивку), затем вручает оператору ЭВМ и ждет результата. Оператор собирает несколько программ, полученных от разных пользователей, и «пропускает» через ЭВМ. При этом ЭВМ, закончив выполнение одной программы, вводит следующую по порядку программу и выполняет ее. Происходит своего рода обработка потока (или пакета) заданий, полученных от разных пользователей.

Крупные ЭВМ используются в мультипрограммном режиме пакетной обработки. Смысл этого режима состоит в том, что в оперативную память ЭВМ поступает сразу несколько программ и выполняются как бы все программы одновременно. Однако это кажущаяся одновременность. На самом деле машина переключается с одной программы на другую, прерывая выполнение одной, переходя ко второй, снова прерывая выполнение, переходя к третьей или продолжая прерванную первую. Таким образом, продвигаются все программы, поступившие для выполнения в мультипрограммном режиме.

Такой режим очень эффективно использует все устройства ЭВМ. Предположим, что первая программа выполнилась до того момента, когда надо вывести на печать ее первые полученные результаты. Вывод на печать по сравнению со счетом — дело медленное, и пока первая программа ждет окончания печати своих результатов, центральный процессор переключается на выполнение второй программы. Следовательно, центральная часть ЭВМ не простаивает в ожидании конца печати. Конечно, для организации мультипрограммирования машина должна быть соответствующим образом устроена. В ней должны одновременно параллельно работать и внешние устройства, и центральная часть. Кроме того, следует позаботиться и о том, чтобы программы, выполняющиеся в мультипрограммном режиме, были защищены от взаимного влияния. Соответственно, в машинах, допускающих мультипрограммный режим, существуют логические электронные схемы, обеспечивающие защиту областей памяти от вмеша-

тельства других программ, обеспечивающие прерывание программ и точный возврат к продолжению их выполнения.

Очень важным элементом математического обеспечения мультипрограммного режима является уже упоминавшаяся ранее операционная система. Это та компонента служебных системных программ, которая обслуживает и управляет одновременной работой нескольких внешних устройств, планирует моменты переключения с программы на программу, осуществляет прерывание программы, возврат к продолжению выполнения прерванных программ. Математическое обеспечение при этом разрабатывается таким образом, чтобы как можно более эффективно использовать аппаратные средства и возможности ЭВМ, не допустить простоев оборудования.

Следующий режим, который мы назвали режимом мультидоступа, часто еще называют режимом многотерминального доступа, режимом дистанционного доступа со многих терминалов. Иногда этот режим классифицируют как режим разделения времени. Действительно, в этом случае вычислительные ресурсы одной ЭВМ делятся между несколькими пользователями, одновременно работающими каждый за своим терминалом. С точки зрения пользователя терминал, снабженный дисплеем и клавиатурой, как бы эквивалентен персональному компьютеру, и взаимодействие такого пользователя с ЭВМ также основано на диалоге с машиной.

Но с точки зрения центральной ЭВМ, к которой подключено несколько десятков терминалов, каждый из них есть внешнее устройство, с которого поступают заказы на выполнение вычислительных работ. Все эти одновременно работающие терминалы следует обслужить как можно скорее, чтобы пользователю, сидящему за терминалом, не приходилось долго ждать ответа от ЭВМ.

Задания и программы, полученные с терминалов, поступают в мультипрограммную обработку, о которой уже шла речь. Но при этом должен быть достигнут компромисс между эффективностью использования всего оборудования ЭВМ и минимумом времени ожидания пользователя. Достичь этого не так просто, и на операционную систему, обеспечивающую многотерминальный доступ, возлагаются сложные задачи планирования обслуживания терминалов.

Для крупных вычислительных машин, к которым подключено много терминалов, характерна комбинация двух режимов — терминального доступа и пакетной обработки. Обслуживание терминалов производится на фоне решения задач в пакетном режиме. При этом приоритет отдается терминалам; в поздние вечерние и ночные часы такие ЭВМ работают в основном в пакетном режиме.

Четвертый режим — режим реального времени — в основном используется на ЭВМ, являющихся основным логическим звеном в различного рода системах управления, например, производственными процессами. Существенным в этом случае является выполнение работ

точно к заданному сроку. Результатом выполнения работ по обработке данных в таких системах является формирование управляющих воздействий и выдача их на объекты управления. В такого рода системах критерий эффективного использования оборудования ЭВМ отступает на задний план. Главное — выдержать точно моменты выдачи управляющих сигналов или команд. Существенную часть обеспечения режима реального времени составляют системные программы, распределяющие все работы по приоритетам, в зависимости от сроков их обязательного завершения. Для того чтобы правильно осуществлять управление, нужно своевременно получать информацию о состоянии управляемых объектов. О своем состоянии эти объекты информируют ЭВМ через систему прерываний. От датчиков в машину в любой момент времени от любого объекта может поступить сигнал: «Обратите на меня внимание», и ЭВМ, работающая в реальном времени, должна прервать все другие работы, с тем чтобы своевременно отреагировать на такого рода сигнал. Следовательно, и в случае режима реального времени возникают те же проблемы в организации прохождения задач, что и в режиме мультипрограммирования: надо быстро прервать выполнение текущей задачи и, если понадобится, переключиться на выполнение другой либо вернуться к продолжению прерванной. В силу этого операционные системы мультипрограммного режима, режима мультидоступа и режима реального времени имеют много общего.

В настоящее время самой дорогой компонентой при разработке вычислительной системы любого класса и назначения является ее программное оборудование. Вот некоторые цифры, подтверждающие этот факт. В СССР над созданием программ трудятся свыше 200 тыс. человек. В США над созданием средств математического обеспечения работают около 250 тыс. человек. Там зарегистрировано 4300 сервисных фирм, осуществляющих разработку, поддержание и поставку математического обеспечения.

Потребность роста числа программистов определяется в 35% в год.

Продажа средств МО растет на 65% в год. Отношение стоимости аппаратуры к стоимости математического обеспечения достигло в настоящее время 1/3. Причем, если стоимость аппаратуры за каждые пять лет снижается приблизительно в 5 раз, то серьезных сдвигов в удешевлении разработки математического обеспечения пока не видно. Был в свое время некий скачок в удешевлении производства программ, связанный с появлением языков высокого уровня, в результате которого стоимость разработки программ в расчете на одну команду резко снизилась (в 3—4 раза).

Разработка математического обеспечения вычислительных систем требует привлечения квалифицированных математиков, программистов и инженеров. Эта одна из самых наукоемких сфер деятельности, имеющая очень важное значение для народного хозяйства, науки, производства.

## ПРИЛОЖЕНИЕ

### ВЫЧИСЛЕНИЯ С ПОМОЩЬЮ МИКРОКАЛЬКУЛЯТОРОВ

#### § П1.1. Общие сведения о микрокалькуляторах

Для решения различных задач вычислительного характера, проведения инженерных расчетов широкое применение находят микрокалькуляторы—портативные электронные вычислительные устройства. В настоящее время у нас в стране и за рубежом производится большое количество вычислительных устройств такого рода, отличающихся как назначением, так и рабочими характеристиками. С учетом конструктивных и функциональных особенностей их можно подразделить на следующие:

- микрокалькуляторы, предназначенные для выполнения четырех основных арифметических операций: сложения, вычитания, умножения, деления;

- микрокалькуляторы, выполняющие обычные арифметические операции и автоматические вычисления некоторых функций ( $\sin x$ ,  $\cos x$ ,  $\ln x$ ,  $\lg x$ ,  $e^x$ ,  $y^x$ ,  $n!$ , ...);

- программируемые микрокалькуляторы, которые обладают всеми возможностями представленных выше микрокалькуляторов, а также могут в автоматическом режиме выполнять последовательность заранее введенных команд.

Каждый микрокалькулятор (МК) независимо от конструкции и функциональных особенностей имеет следующие составные части:

- клавиатуру—набор кнопок для ввода данных и операций над ними;

- световой индикатор для чтения результата и операндов;

- счетно-решающее устройство, выполняющее вычислительные операции.

Основные связи между составными частями МК представлены на рис. П1.

Многие типы МК имеют в своем составе дополнительную память (на рис. П1 она обведена пунктиром). Ее назначение—хранить резуль-



таты промежуточных вычислений, а также для программируемых МК—программу.

В МК возможно представление чисел как с фиксированной, так и с плавающей запятой (представления были рассмотрены в гл. 3).

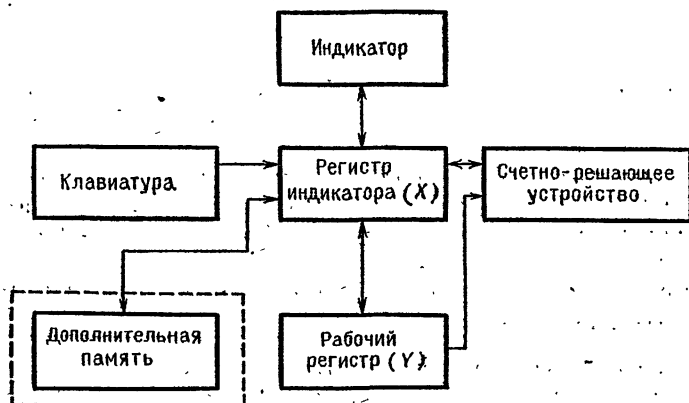


Рис. П1

Под изображение цифр числа в каждой модели МК отводится фиксированное количество разрядов. Существуют шести-, восьми-, и десятиразрядные МК.

Например, число —12,789601 на индикаторе восьмиразрядного МК «Электроника БЗ-36» выглядит следующим образом:



Как известно, количество разрядов, отводимых под изображение числа, определяет диапазоны и точность представления чисел. Легко видеть, что в шестиразрядном МК, работающем с величинами, представленными в форме с фиксированной запятой, могут быть представлены отличные от нуля числа, абсолютные значения которых лежат в интервале  $0,00001 \div 99999$ . В восьми- и десятиразрядных МК мы можем работать соответственно с числами из интервалов  $0,0000001 \div 99999999$  и  $0,000000001 \div 9999999999$ . Во всех случаях, когда мы попытаемся ввести число, требующее для своего изображения большее количество разрядов, чем это определено конструкцией МК, на индикаторе получим только старшие разряды этого числа. Например, если в восьмиразрядный МК попытаться ввести число 900116589, то на его индикаторе получим 90011658, которое отличается от необходимого нам. Разумеется, использование этого числа при вычислениях даст нам неверный результат.

Аналогичная ситуация происходит и в том случае, когда в результате арифметических операций получается число, значащих разрядов в изображении которого больше, чем предусмотрено конструкцией МК. В этом случае мы приходим к ситуации, называемой переполнением, например,

$$888888 \times 246795 = 219373113960$$

При попытке умножить эти два числа при помощи восьмиразрядного МК мы на индикаторе получим число 21937311 и в знаковом разряде светящуюся точку — признак переполнения. С другой стороны, любое число, модуль которого меньше левой границы из диапазона представления для данного МК, является в нем нулем. Например, равными нулю в шестиразрядном МК будут числа 0.0000011; 0.000009.

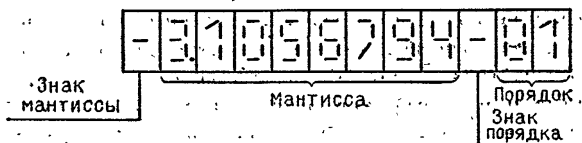
В результате вычисления на шестиразрядном МК произведения  $0.00009 \times 0.1 = 0.000009$

мы получим на индикаторе нулевое значение.

В некоторых моделях МК наряду с представлением чисел с фиксированной запятой используется также представление чисел в форме с плавающей запятой.

Как известно, для заданий числа в форме с плавающей запятой нужно указать мантиссу и порядок. Под изображение мантиссы, как правило, отводятся шесть или восемь разрядов, под изображение порядка во всех моделях — два разряда.

Мантисса и порядок числа имеют знак. Причем знак самого числа определяется знаком мантиссы. Для изображения знаков мантиссы и порядка отводятся специальные разряды. Например, число  $-0.31056794$  на индикаторе МК в форме с плавающей запятой выглядит следующим образом:



Следует отметить, что МК при работе с числами в форме с плавающей запятой выдает их, как правило, таким образом, что запятая у этих чисел находится после первого, отличного от нуля знака мантиссы.

Возможность работы с числами, представленными в форме с плавающей запятой, существенным образом расширяет диапазон чисел, с которыми может работать МК. Так, например, для случая шестизначной мантиссы и положения десятичной запятой после первого знака мантиссы МК может работать с числами, абсолютная величина

которых лежит в диапазоне

$$10^{-99} - 9.99999 \times 10^{99}$$

Если для изображения мантиссы отводится восемь знаков, то правая граница диапазона соответственно расширяется до  $9.9999999 \times 10^{99}$ .

В МК имеются клавиши ввода 0, 1, ..., 9, предназначенные для ввода цифр от 0 до 9 и запятой (в некоторых моделях — точки), отделяющей целую часть числа от дробной. Если МК работает с числами, представленными в форме с плавающей запятой, то в нем для ввода порядка числа имеется специальная клавиша. Например, либо EXP, либо EE, либо VP.

Клавиши, на которых обозначены четыре основных арифметических действия, т. е.  $+$   $-$   $\times$   $\div$ , а также клавиша  $=$  — вывода результата (ею, как правило, завершают последовательность операций) относятся к группе *операционных клавиш*.

Клавиша  $/-/$  позволяет изменять знак числа, представленного на индикаторе, на противоположный. Ею, например, пользуются при вводе в МК отрицательных чисел.

Следует отметить, что независимо от формы представления чисел в МК при их вводе каждое число набирается при помощи клавиш ввода последовательно, цифра за цифрой, вместе с запятой (точкой), отделяющей целую часть от дробной, начиная со старших разрядов.

Например, для того чтобы ввести число 20,053, необходимо последовательно нажать клавиши

2 0 , 0 5 3

Порядок заполнения экрана цифрами зависит от модели МК. Например, он может быть таким:

Ввод	Показания индикатора	Ввод	Показания индикатора
2	2	0	20.0
0	20	5	20.05
.	20.	3	20.053

Числа, представленные в форме с плавающей запятой, вводятся следующим образом. Сначала цифра за цифрой вводится мантисса, а затем порядок числа.

Причем после ввода мантиссы необходимо нажать специальную клавишу для ввода порядка. Сигналом для МК об окончании ввода может быть, например, нажатие одной из операционных клавиш.

Для выполнения какой-либо арифметической операции над двумя числами в большинстве моделей МК необходимо выполнить следующую последовательность действий:

- ввести первое число,
- задать требуемую операцию, нажав одну из операционных клавиш,
- ввести второе число,
- вывести результат, нажав клавишу  $=$ .

Вычислим произведение двух чисел  $20.1 \times 3.2$ , например, при помощи МК «Электроника БЗ-18М».

Для этого необходимо нажать на его клавиатуре следующие клавиши в представленной ниже последовательности:

$2 \ 0 \ . \ 1 \ \times \ 3 \ . \ 2 \ =$

На индикаторе при выполнении этих манипуляций будут появляться следующие последовательности знаков:

Ввод	Показания индикатора	Ввод	Показания индикатора
2	2	3	3
0	20	.	3.
.	20.	2	3.2
. 1	20.1	=	64.32
$\times$	20.1		

Рассмотрим, как происходит перемещение данных в МК. Напомним, что МК имеет в своем распоряжении регистр индикатора X и рабочий регистр Y (см. рис. П1). В выполнении арифметических операций участвуют оба эти регистра. При вводе первого операнда это число размещается в регистре X и высвечивается на индикаторе. После нажатия одной из операционных клавиш содержание регистра X пересылается в регистр Y. При этом содержимое регистра X остается неизменным, т. е. в регистр Y помещается как бы дубль того, что хранится в регистре X. После ввода следующего числа, участвующего в операции, оно помещается в регистр X, стирая при этом ранее занесенный туда первый операнд. Таким образом, микрокалькулятор располагает обоими операндами (первый находится в регистре Y, а второй в регистре X и одновременно он высвечивается на индикаторе), и информацией о том, какую операцию над ними необходимо выполнить. При нажатии на клавишу  $=$  происходит выполнение подготовленной операции над содержимыми регистров X и Y.

Результат помещается в регистр X и высвечивается на индикаторе. Второй операнд при этом пересылается из регистра X в регистр Y, прежнее содержимое которого стирается.

Пусть, например, нам нужно вычислить разность двух чисел  $21.03 - 1.02$ . Для этого достаточно последовательно нажать клавиши:

$2 \ 1 \ . \ 0 \ 3 \ - \ 1 \ . \ 0 \ 2 \ =$

Покажем, как при этом меняется содержимое регистров X и Y:

Ввод	Содержимое X	Содержимое Y	Ввод	Содержимое X	Содержимое Y
2	2	0	1	1	21.03
1	21	0	.	1.	21.03
0	21.	0	0	1.0	21.03
.	21.0	0	2	1.02	21.03
3	21.03	0	=	20.01	1.02
—	21.03	21.03			

Многие модели МК имеют клавишу  $\leftrightarrow$ , назначение которой состоит в том, чтобы производить обмен содержимым между регистрами X и Y. Действительно, если к моменту нажатия клавиши  $\leftrightarrow$  содержимое регистра X равнялось 0.54, а в регистре Y находилось число 22, то после нажатия этой клавиши будем иметь в регистре Y число 0.54, а в регистре X и, следовательно, на индикаторе — число 22.

Эта клавиша оказывается полезной как при выполнении арифметических операций (необходимо помнить о том, что для каждой из них в качестве первого операнда берется содержимое регистра Y, а в качестве второго — содержимое X), так и для наблюдений за содержимым регистров X и Y в процессе вычислений.

Как уже отмечалось, во многих моделях МК реализована возможность вычисления значений некоторых элементарных функций по заданному значению аргумента.

Набор функций, вычисление которых реализовано в той или иной модели МК, представлен на его клавиатуре. Современная технология производства интегральных схем позволяет сделать миниатюрное вычислительное устройство с достаточно широкими вычислительными возможностями. Все это дает возможность увеличить число различных функций, для вычисления значений которых достаточно нажать соответствующую клавишу на клавиатуре МК. Однако с ростом числа клавиш мы обязаны увеличивать и размеры самого МК, так как идти по пути уменьшения размеров клавиш мы можем только до определенных пределов — если они слишком маленькие, то с ними трудно работать. В связи с этим во многих моделях МК некоторые клавиши могут быть использованы для различных целей. При этом указатель на первое функциональное назначение такой клавиши наносится непосредственно на ней самой, а указатели на другие функциональные назначения наносятся на корпус МК в непосредственной близости от этой клавиши. Если нужно вычислить значение функции, название которой обозначено на клавише, то достаточно нажать саму эту клавишу. Для вычислений значений функций, обозначение кото-

рых нанесено на корпус МК, нужно вначале нажать клавишу F, а затем клавишу нужной функции.

В качестве значения аргумента вычисляемой функции выступает содержимое регистра индикатора X, т. е. перед вычислением значения любой функции нужно ввести аргумент, а затем нажать клавишу, соответствующую вычисляемой функции.

Результат также помещается в регистр X, содержимое которого при этом, т. е. аргумент вычисленной функции, переходит в регистр Y.

Например, для вычисления значения  $\lg 20$  на МК «Электроника БЗ-18М» достаточно выполнить следующую последовательность действий:

2 0 F lg

Содержимое регистров, при этом будет меняться следующим образом:

Ввод	Регистр X	Регистр Y
2	2	0
0	20	0
F	20	0
lg	1.30103	20

При вычислении значений тригонометрических функций почти во всех моделях МК предусмотрена возможность задавать аргумент как в градусах, так и в радианах (для перехода от градусов к радианам предусмотрен специальный переключатель).

Вычислим, например,  $\cos(-35^\circ)$  на МК «Электроника БЗ-18М». Для этого поставим переключатель ГРАД—РАД в положение ГРАД и нажмем на панели МК следующие клавиши:

3 5 /—/ F cos

На индикаторе МК при этом будем иметь:

Ввод	Индикатор, регистр X
3	3
5	35
/—/	—35
F	—35
cos	0.819152

Существуют и такие модели, где аргумент задается либо в градусах, либо в радианах. В этом случае для перехода от градусов к радианам и обратно следует пользоваться известными формулами:

$$\alpha_{\text{град}} = \frac{180^\circ}{\pi} \cdot \alpha_{\text{рад}}; \quad \alpha_{\text{рад}} = \frac{\pi}{180^\circ} \cdot \alpha_{\text{град}}.$$

Здесь же отметим, что при вычислении значения функции МК выдает правильный результат в том случае, если аргумент принадлежит области ее определения. Например, квадратные корни можно извлекать лишь из неотрицательных чисел, логарифмы определены лишь для положительных значений аргумента. Некоторые модели МК снабжены системой контроля допустимости значения аргумента, и если он не попадает в область определения вычисляемой функции, то на индикаторе выдается соответствующий сигнал. Однако такая возможность реализована не везде. Если система контроля отсутствует, то в этом случае при вычислениях необходимо самому контролировать правильность задания аргумента.

Для лучшего понимания возможностей МК опишем кратко имеющиеся в нем регистры и предоставляемые ими возможности. Основное назначение регистров — хранить числа, используемые в качестве операндов в различных операциях. Минимально возможное число таких регистров равняется двум. Это уже рассмотренные нами ранее регистр индикатора X и рабочий регистр Y. Кроме того, во многих моделях имеются специальные регистры памяти, которые обеспечивают значительные удобства при проведении вычислений. Основное их назначение — хранить результаты промежуточных вычислений и часто используемые константы. Ведь если такой возможности нет, то мы вынуждены записывать эти результаты на бумаге либо запоминать их, и по мере надобности вводить заново в МК.

Простейшие МК имеют, как правило, не более одного регистра памяти. Однако существуют модели, которые могут иметь достаточно большое число таких регистров.

Регистры памяти бывают двух типов: простые регистры и регистры, накапливающие результат. Для управления содержимым простого регистра памяти на клавиатуре МК имеется пара специальных клавиш, позволяющих пересылать содержимое регистра индикатора X в регистр памяти П, а также содержимое регистра памяти П в регистр индикатора X. Здесь следует подчеркнуть, что содержимое регистра, из которого пересылается число, сохраняется неизменным. Число же, находившееся в регистре, в который осуществляется пересылка, стирается и заменяется новым.

Например, после нажатия последовательности клавиш

0 . 2 5 F П

на панели МК на его индикаторе и в регистрах мы будем иметь:

Ввод	Регистр X	Регистр Y	Регистр П
0.25	0.25	0	0
F	0.25	0	0
П	0.25	0	0.25

Пусть в какой-то момент времени регистры МК содержали следующие числа:

Регистр X	Регистр Y	Регистр П
100	20.1	0.63

Тогда после нажатия клавиши  $\text{П} \rightarrow \text{X}$  получим:

Регистр X	Регистр Y	Регистр П
0.63	20.1	0.63

Значительные удобства при работе с МК обеспечивает клавиша  $\text{X} \leftrightarrow \text{П}$ , позволяющая производить обмен между содержанием регистра индикатора X и регистра памяти П.

Пусть, как в рассмотренном ранее примере в регистрах МК находились следующие числа:

Регистр X	Регистр Y	Регистр П
100	20.1	0.63

После нажатия клавиши  $\text{X} \leftrightarrow \text{П}$  получим:

Регистр X	Регистр Y	Регистр П
0.63	20.1	100

Содержимое регистра Y при этом осталось неизменным. Как уже отмечалось, некоторые модели МК в качестве регистров памяти имеют



так называемые накапливающие регистры. Существенное отличие такого регистра от простого регистра памяти заключается в том, что при выполнении операции засылки в такой регистр, туда попадает результат одной из арифметических операций, где операндами выступают содержимое этого регистра памяти и регистра X.

Наличие у МК накапливающих регистров можно определить по клавишам  $\Pi+$ ,  $\Pi-$ ,  $\Pi\times$ ,  $\Pi\div$ . После нажатия клавиши  $\Pi+$  в регистре

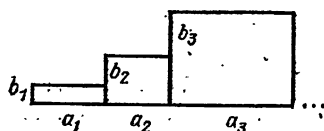


Рис. П2

памяти окажется сумма двух чисел: числа, хранившегося в регистре памяти и числа, находившегося в регистре X. При нажатии клавиши  $\Pi-$  в регистре памяти окажется разность между числом, хранившимся в нем, и числом, находящемся в регистре X.

В результате нажатия клавиши  $\Pi\times$  ( $\Pi\div$ ) прежнее содержимое регистра памяти умножится (разделится) на содержимое регистра X. Результат операции будет помещен в регистр памяти.

В качестве примера использования накапливающего регистра рассмотрим вычисление суммы произведений:

$$S = \sum_{i=1}^n a_i \cdot b_i$$

Выражение такого вида может, например, определять площадь фигуры, состоящей из  $n$  прямоугольников со сторонами  $a_i$ ,  $b_i$  ( $i = 1, 2, \dots, n$ ).

Пусть  $n=3$ ,  $a_1=5$ ,  $a_2=4$ ,  $a_3=8$  и  $b_1=2$ ,  $b_2=3$ ,  $b_3=6$ , т. е. фигура на плоскости может выглядеть следующим образом (рис. П2).

С использованием накапливающего регистра памяти площадь фигуры можно вычислить следующим образом:

$$5 \times 2 = \Pi+ \quad 4 \times 3 = \Pi+ \quad 8 \times 6 = \Pi+$$

Содержание регистра X и регистра  $\Pi$  при этом будут изменяться следующим образом:

Ввод	Регистр X	Регистр $\Pi$	Ввод	Регистр X	Регистр $\Pi$
5	5	0	=	12	10
$\times$	5	0	$\Pi+$	12	22
2	2	0	8	8	22
=	10	0	$\times$	8	22
$\Pi+$	10	10	6	6	22
4	4	10	=	48	22
$\times$	4	10	$\Pi+$	48	70
3	3	10			

Теперь для того, чтобы получить результат на экране индикатора, достаточно нажать клавишу  $X \leftrightarrow P$ .

У МК, имеющих в своем составе регистры памяти, на клавиатуре располагается, как правило, клавиша СП, позволяющая стирать содержание регистра памяти. Если такой клавиши нет, то очистку регистра памяти можно выполнить, заслав в него нуль.

## § П2. Некоторые приемы решения задач на простейших микрокалькуляторах

Выше мы познакомились с некоторыми общими сведениями об МК, их конструктивными особенностями. Рассмотрим теперь некоторые приемы решения задач с использованием МК основных типов, классификация которых приведена в § П1.

Остановимся вначале на особенностях работы с МК, выполняющим четыре основных арифметических действия и не обладающим регистрами памяти. Типичный МК этого семейства имеет вид, представленный на рис. П3. Обычно такие МК выполняют действия над числами, представленными в форме с фиксированной запятой. Индикатор в большинстве моделей шести- или восьмиразрядный. Довольно часто в таких МК клавиша выдачи результатов совмещена с клавишами  $+$  и  $-$ .

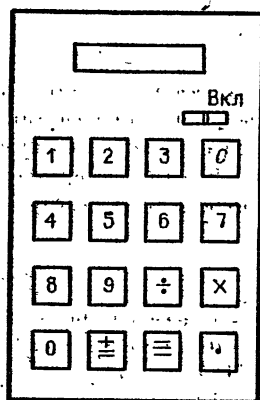


Рис. П3

или  $-$ , т. е. имеются клавиши  $=$  или  $\neq$ .

Например, для вычисления произведения двух чисел 2,83 и 6,792 достаточно выполнить следующие действия:

$$2.83 \times 6.792 =$$

В результате на индикаторе шестиразрядного МК получим 19.2214. На индикаторе восьмиразрядного МК будем иметь 19.22136. Отсюда видно, что МК автоматически производит округление результата.

На простейших МК удобно выполнять последовательность нескольких идущих подряд арифметических действий. Например, вычислим выражение:

$$1.45 - 0.849 + 10.036$$

Порядок действий может, например, быть таким:

$$1.45 - 0.849 = 10.036 =$$

При этом содержимое индикатора МК будет изменяться в следующей последовательности:

Ввод	Индикатор	Ввод	Индикатор
1.45	1.45	$\begin{smallmatrix} + \\ = \end{smallmatrix}$	0.601
$\begin{smallmatrix} - \\ = \end{smallmatrix}$	1.45	10.036	10.036
0.849	0.849	$\begin{smallmatrix} + \\ = \end{smallmatrix}$	10.637

**Пример.** Пусть необходимо вычислить площадь трапеции с основаниями  $a=3$ ;  $b=5$  и высотой  $h=1.57$ . Известно, что  $S=(a+b) \times h/2$ . В нашем случае  $S=(3+5) \cdot 1.57/2$ .

Для решения задачи на МК достаточно нажать клавиши в следующем порядке:

$$3 \begin{smallmatrix} + \\ = \end{smallmatrix} 5 \times 1.57 \div 2 \begin{smallmatrix} + \\ = \end{smallmatrix}$$

В результате получим  $S=6.28$ .

В этом примере встретилось выражение вида  $(a+b) \cdot h$ . При его вычислении после ввода числа  $a$  и нажатия клавиши  $\begin{smallmatrix} + \\ = \end{smallmatrix}$ , затем ввода второго операнда  $b$  МК готов к выполнению заданной операции. Нажатие операционной клавиши  $\times$  производит установленную ранее операцию сложения и подготавливает МК к выполнению операции умножения. Первым операндом операции умножения выступает результат, полученный на предыдущем шаге (а именно, сумма  $(a+b)$ ), второй операнд (значение  $h$ ) мы вводим при помощи клавиатуры. Теперь нам необходимо выполнить операцию умножения и подготовить МК к выполнению следующей операции — деления на 2. Для этого достаточно нажать клавишу  $\div$ , что приведет к выполнению уже подготовленной операции умножения, результат которой явится первым операндом операции деления. Далее нужно ввести делитель, в нашем случае это число 2, и вновь нажать одну из операционных клавиш, например  $\begin{smallmatrix} + \\ = \end{smallmatrix}$ . Выполнится операция деления и результат высветится на индикаторе.

**Пример.** Вычислить

$$S = 6.9 \times 5.3 \times (5.7 - 0.234 + 1/6)$$

с использованием восьмиразрядного МК.

Легко видеть, что существует несколько способов решения поставленной задачи. Можно, например, это сделать так:

Ввод	Индикатор	Примечания	Ввод	Индикатор	Примечания
6.9	6.9		1	1	
×	6.9		÷	1	
5.3	5.3		6	6	
=	36.57	Записать на бумаге первый промежуточный результат	+	0.1666666	
			5.466	5.466	
			×	5.6326666	
			36.57	36.57	
5.7	5.7		=	205.98661	
—	5.7				
0.234	0.234				
=	5.466	Записать на бумаге второй промежуточный результат			

В результате получим  $S=205,98661$ . В этом случае нам дважды приходилось записывать на бумаге или фиксировать в собственной памяти результаты промежуточных вычислений, а затем вновь их вводить в МК.

Очевидно, что рациональнее при нахождении значения  $S$  было бы поступить следующим образом: вначале вычислить значение выражения, стоящего в скобках, а затем дважды выполнить операцию умножения, т. е. проделать следующую последовательность действий:

$$1 \div 6 - 0.234 + 5.7 \times 5.3 \times 6.9 =$$

Ход решения:

Ввод	Показания индикатора	Ввод	Показания индикатора
1	1	5.7	5.7
÷	1	×	5.632667
6	6	5.3	5.3
—	0.1666666	×	29.853134
0.234	0.234	6.9	6.9
+	-0.0673333	=	205.98661

Здесь мы уже обошлись без фиксирования в памяти или на бумаге промежуточных результатов и, более того, проделали меньшее количество манипуляций клавишами. Таким образом при вычислении значения выражения рассмотренного в нашем примере типа целесообразно на первом этапе вычислить выражение в скобках, а затем произвести операцию умножения.

Пример. Вычислить значение многочлена

$$P_5(x) = x^5 + 3x^4 - 4x^3 + 2x^2 + 5x - 8$$

при  $x = 2,1$ .

Решая задачу «в лоб», мы должны вычислить последовательно  $2,1^5$ ,  $3 \times 2,1^4$ ,  $4 \times 2,1^3$ ,  $2 \times 2,1^2$ ,  $5 \times 2,1$ , зафиксировать эти значения в качестве промежуточных результатов, а затем выполнить над ними операции сложения и вычитания.

Гораздо эффективнее в этом случае применить схему Горнера (о ней мы уже говорили выше), согласно которой

$P_5(x) = x^5 + 3x^4 - 4x^3 + 2x^2 + 5x - 8 = (((x+3)x-4)x+2)x+5)x-8$ , т. е. при  $x = 2,1$  нужно вычислить значение выражения

$$(((2,1+3) \times 2,1 - 4) \times 2,1 + 2) \times 2,1 + 5) \times 2,1 - 8.$$

Решение задачи в этом случае может выглядеть следующим образом:

Ввод	Показания индикатора	Ввод	Показания индикатора
2.1	2.1	2	2
+	2.1	×	16.091
3	3	2.1	2.1
×	5.1	+	33.7911
2.1	2.1	5	5
-	10.71	×	38.7911
4	4	2.1	2.1
×	6.71	-	81.46131
2.1	2.1	8	8
+	14.091	=	73.46131

Здесь нам удалось обойтись без запоминания и повторного ввода в МК промежуточных результатов.

Очевидно, что для вычисления многочлена

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 =$$

$$= (\dots ((a_n x + a_{n-1}) x + a_{n-2}) x + \dots + a_1) x + a_0$$

произвольной степени  $n$  по схеме Горнера достаточно нажать клавиши МК в таком порядке

$$a_n \times x + a_{n-1} \times x + a_{n-2} \times \dots \times x + a_0 =,$$

где  $a_i$  ( $i = 0, 1, 2, \dots, n$ ) — коэффициенты многочлена.

Мы уже говорили, что при решении различных задач часто возникает необходимость в вычислении суммы вида

$$S = \sum_{i=1}^n a_i b_i$$

Пусть, например, надо решить задачу.

Расстояние между двумя городами поезд прошел по следующему графику: первые полтора часа он ехал со скоростью 45 км/ч, затем два часа со скоростью 46,8 км/ч, потом 4 ч со скоростью 53,5 км/ч и 3 ч со скоростью 50,8 км/ч. Каково расстояние между двумя городами?

Очевидно, что  $S = v_1 t_1 + v_2 t_2 + v_3 t_3 + v_4 t_4 = 45 \times 1,5 + 46,8 \times 2 + 53,5 \times 4 + 50,8 \times 3$ .

Если теперь попытаться вычислить каждое слагаемое и потом их просуммировать, то мы будем вынуждены фиксировать на бумаге три промежуточных результата.

Легко видеть, что выражение для  $S$  можно привести к виду

$$S = \left( \left( \left( \frac{45}{46,8} \cdot 1,5 + 2 \right) \cdot \frac{46,8}{53,5} + 4 \right) \cdot \frac{53,5}{50,8} + 3 \right) \cdot 50,8.$$

Теперь уже мы можем вычислить значение  $S$  без записи промежуточных результатов. Однако, если при первом способе вычислений нам было достаточно выполнить семь арифметических операций, то при втором способе пришлось выполнить уже десять операций (добавилось три операции деления):

Ввод	Индикатор	Ввод	Индикатор
45	45	+	3.0112147
÷	45	4	4
46.8	46.8	×	7.0112147
×	0.9615385	53.5	53.5
1.5	1.5	÷	375.09998
+	1.4423076	50.8	50.8
2	2	+	7.3838578
×	3.4423076	3	3
46.8	46.8	×	10.383857
÷	161.09999	50.8	50.8
53.5	53.5	=	527.49993

Заметим, что в общем случае выражение

$$S = \sum_{i=1}^n a_i b_i = a_1 b_1 + a_2 b_2 + \dots + a_n b_n \quad \text{при} \quad a_i \neq 0, \quad i = 2, 3, \dots, n,$$

приводится к виду

$$S = \left( \left( \dots \left( \frac{a_1 b_1}{a_2} + b_2 \right) \frac{a_2}{a_3} + b_3 \right) \frac{a_3}{a_4} + b_4 \right) \frac{a_4}{a_5} + \dots + b_{n-1} \right) \times \left( \frac{a_{n-1}}{a_n} + b_n \right) a_n.$$

Для вычисления значения  $S$  в этом случае достаточно нажать

клавиши МК в следующем порядке:

$$a_1 \times b_1 \div a_2 + b_2 \times a_2 \div a_3 + b_3 \times \dots + b_{n-1} \times a_{n-1} \div a_n + b_n \times a_n =$$

Пример. В прямоугольном треугольнике найти гипотенузу  $c$  по двум катетам  $a=2.1$  и  $b=3.8$ . Известно, что  $c=\sqrt{a^2+b^2}$ . В нашем случае  $c=\sqrt{2.1^2+3.8^2}$ .

Подкоренное выражение вычисляется просто и оно равно 18.85. Значение квадратного корня из 18.85 можно взять, например, из таблиц. Однако не так сложно эта задача решается и при помощи простейшего МК.

Например, для этой цели можно использовать итерационный метод Ньютона, который позволяет вычислять значение  $\sqrt{a}$ ,  $a \geq 0$ , с некоторой наперед заданной точностью  $\epsilon$ . Как известно (см. гл. 2), последовательное приближение к точному значению корня осуществляется по формуле

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right), \quad n=1, 2, 3, 4, \dots$$

Найдем значение  $\sqrt{18.85}$ . В качестве первого приближения возьмем  $x_1=4$ . Тогда

Ввод	Показания индикатора	Примечания
18.85	18.85	1-я итерация
$\div$	18.85	
4	4	
$+$	4.7125	
4	4	
$\div$	8.7125	2-я итерация
2	2	
$=$	4.35625	
18.85	18.85	
$\div$	18.85	
4.35625	4.35625	3-я итерация
$+$	4.3271162	
4.35625	4.35625	
$\div$	8.6833662	
2	2	
$=$	4.3416831	3-я итерация
18.85	18.85	
$\div$	18.85	
4.3416831	4.3416831	
$+$	4.3416342	
4.3416831	4.3416831	3-я итерация
$\div$	8.6833173	
2	2	
$=$	4.3416586	

Легко видеть, что значение  $x_4$  отличается от  $x_3$  в пятом знаке после запятой. Таким образом уже после трех итераций мы получили значение  $\sqrt{18.85}$  с точностью  $\varepsilon = 10^{-4}$ .

Довольно часто при решении задач элементарной математики возникает необходимость в вычислении значений элементарных функций. Обычно в этом случае пользуются таблицами значений этих функций. Как же решить эту задачу, если под рукой нет таблиц, а есть простейший МК?

Напомним (см. гл. 2), что многие элементарные функции можно представить в виде некоторых бесконечных сумм. Например, для  $\sin x$  и  $\cos x$  эти представления выглядят так

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + (-1)^n \frac{x^{2n+1}}{(2n+1)!} \pm \dots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + (-1)^n \frac{x^{2n}}{(2n)!} \pm \dots$$

Здесь  $x$  — значение угла в радианной мере.

На первом этапе вычисления значений этих функций необходимо позаботиться о том, чтобы аргумент  $x$  был представлен в радианной мере. Если МК не снабжен специальным переключателем РАД — ГРАД, то можно воспользоваться приведенным на с. 278 формулами для перехода от градусной меры угла к радианной. Вычисления по этим формулам легко выполнить с помощью простейшего МК:

$$x_{\text{гр}} \div 180 \times 3,1415926 =$$

В результате на индикаторе МК получим значение  $x$  в радианах.

Вновь вернемся к выражениям для  $\sin x$ ,  $\cos x$ . Обратим внимание на тот факт, что каждый следующий член в разложении получается из предыдущего по простому алгоритму. Например, члены разложения  $\sin x$  связаны между собой следующим образом:

$$x \cdot x : 2 \cdot x : 3 \cdot x : 4 \cdot x : 5 \cdot x : 6 \cdot x : 7 \cdot x : 8 \dots$$

1-й член

2-й член

3-й член

4-й член

Для членов степенного разложения функции  $\cos x$  справедливо:

$$x \cdot x : 2 \cdot x : 3 \cdot x : 4 \cdot x : 5 \cdot x : 6 \cdot x : 7 \dots$$

2-й член

3-й член

4-й член



Запишем последовательность нажатия клавиш МК для вычисления значений слагаемых в разложении функции  $\sin x$ :

$$x \times x \div 2 \times x \div 3 \times$$

(записать значение 2-го члена разложения)

$$x \div 4 \times x \div 5 \times$$

(записать значение 3-го члена разложения)  $x \dots$

Для вычисления членов разложения  $\cos x$ :

$$x \times x \div 2 \times$$

(записать значение 2-го члена разложения)

$$x \div 3 \times x \div 4 \times$$

(записать значение 3-го члена разложения)

$$x \div 5 \times x \div 6 \times$$

(записать значение 4-го члена разложения)

$$x \div 7 \times \dots$$

Процесс вычисления необходимо продолжать до тех пор, пока абсолютная величина очередного члена разложения не станет меньше, чем некоторое заданное  $\varepsilon > 0$ , определяющее точность вычисления функций. Это обязательно произойдет на каком-либо шаге алгоритмического процесса, так как очевидно, что в разложении  $\sin x$  и  $\cos x$  с ростом  $n$  абсолютные значения слагаемых стремятся к нулю.

Причем процесс этот будет происходить тем быстрее, чем меньше по модулю значение  $x$ . В этом случае для получения удовлетворительного приближения достаточно взять всего лишь несколько первых членов разложения. Для уменьшения значения аргумента  $x$  полезно воспользоваться известными соотношениями:

$$\sin x = \cos(90^\circ - x),$$

$$\cos x = \sin(90^\circ - x).$$

Вновь вернемся к вычислению значений функций  $\sin x$  и  $\cos x$ . После того как получен набор членов разложения, последний из которых меньше заданного  $\varepsilon > 0$ , определяющего требуемую точность вычисления функции, остается только их просуммировать с учетом знаков, с которыми они входят в разложение.

Для функции  $\sin x$ , первый член разложения которого равен  $x$ , имеем:

$$x - 2\text{-й член} + 3\text{-й член} - 4\text{-й член} + \dots =$$

Аналогично для  $\cos x$ , где первый член в разложении равен единице:

$$1 - 2\text{-й член} + 3\text{-й член} - 4\text{-й член} + \dots =$$

Зная значения  $\sin x$  и  $\cos x$ , можно получить и значения других тригонометрических функций, например,  $\operatorname{tg} x$ ,  $\operatorname{ctg} x$ . Аналогичный под-

ход можно использовать при вычислениях на МК значений  $e^x$ ,  $\ln x$  ( $x > 0$ ).

В гл. 2 рассматривалась задача исследования и нахождения решения системы двух линейных алгебраических уравнений с двумя неизвестными

$$\begin{cases} a_{11}x + a_{12}y = b_1, \\ a_{21}x + a_{22}y = b_2. \end{cases}$$

Здесь  $a_{ij}$ ,  $b_j$  ( $i, j=1, 2$ ) — заданные коэффициенты. Если  $a_{11}a_{22} - a_{12}a_{21} \neq 0$ , то решение системы, как мы знаем, выражается формулами:

$$x = \frac{a_{22} \cdot b_1 - a_{12} \cdot b_2}{a_{11} \cdot a_{22} - a_{12} \cdot a_{21}}, \quad y = \frac{a_{11} \cdot b_2 - a_{21} \cdot b_1}{a_{11} \cdot a_{22} - a_{12} \cdot a_{21}}.$$

Для того чтобы использовать изложенный ранее прием вычисления на МК, выражения для  $x$  и  $y$  следует переписать следующим образом:

$$x = \frac{(a_{22}b_1/b_2 - a_{12}) b_2}{(a_{11}a_{22}/a_{12} - a_{21}) a_{12}}, \quad y = \frac{(a_{11}b_2/b_1 - a_{21}) b_1}{(a_{11}a_{22}/a_{12} - a_{21}) a_{12}}.$$

При вычислении значений  $x$  и  $y$  вначале вычислим знаменатель, а затем числитель каждого выражения. Процесс решения задачи может выглядеть следующим образом:

1) вычисление знаменателя

$$a_{11} \times a_{22} \div a_{12} - a_{21} \times a_{12} =$$

Если в результате на индикаторе МК получим нуль, то это может означать, что наша система либо не имеет решения, либо решений бесконечно много. Полученный результат необходимо записать для запоминания на бумаге.

2) вычисление  $x$

$$a_{22} \times b_1 \div b_2 - a_{12} \times b_2 \div \text{значение знаменателя} =$$

3) вычисление  $y$

$$a_{11} \times b_2 \div b_1 - a_{21} \times b_1 \div \text{значение знаменателя} =$$

Предлагаем читателю, следуя рассмотренной выше последовательности действий, найти решения системы

$$\begin{cases} 3x + 2y = 5, \\ 7x + 4y = 2. \end{cases}$$

Ответ.  $x = -8$ ,  $y = 14.5$ .

В настоящем параграфе мы познакомились с некоторыми приемами решения вычислительных задач с использованием МК, выполняющего только четыре основные арифметические операции. Мы имели возможность убедиться в том, что набор задач, решение которых под силу простейшему МК, достаточно широк. Однако, как правило, для эффективного их решения нам приходилось приводить эти задачи к виду,

удобному для решения на МК, учитывающему его возможности. Необходимо подчеркнуть, что в любом случае перед тем, как начать вычисления, следует внимательно проанализировать задачу, определить оптимальный алгоритм ее решения с учетом возможностей имеющегося в распоряжении вычислительного устройства.

Как уже отмечалось ранее, имеются модели МК, которые, наряду с выполнением основных арифметических операций, могут вычислять значения элементарных функций, таких как  $\sin x$ ,  $\cos x$ ,  $\sqrt{a}$  ( $a > 0$ ),  $\ln x$  ( $x > 0$ ) и многих других. Некоторые из них обладают при этом одним или несколькими специальными регистрами памяти. Наличие клавиш для вычисления различных функций, а также регистров памяти значительно облегчают вычисления на МК. Теперь уже для вычисления значения функции достаточно задать значение аргумента и нажать клавишу, соответствующую этой функции. Регистр памяти также создает удобства, так как при его наличии часто отпадает необходимость в фиксации на бумаге промежуточных результатов. Для того, чтобы переслать число в память из регистра ввода  $X$  и наоборот — вызвать число в регистр  $X$  из регистра памяти — достаточно нажать только одну клавишу (или две, если в МК реализован режим совмещения функций клавиш засылки в память и чтения).

Например, пусть необходимо вычислить выражение вида

$$S = (a + bcd)/(ek - lf).$$

Можно в данном случае поступить следующим образом. Вычислить знаменатель, зафиксировать его значение на бумаге. Затем вычислить числитель и разделить его на знаменатель. При этом пришлось бы полученное значение знаменателя вводить в МК. Вычисляя знаменатель, можно воспользоваться рассмотренным выше приемом и переписать его так:

$$ek - lf = (ek/l - f) l.$$

В этом случае можно обойтись без фиксации значений отдельных слагаемых знаменателя (либо  $ek$ , либо  $lf$ ).

Однако, если в МК есть регистр памяти, то можно таких преобразований и не делать, а полученное значение одного из слагаемых сохранить в этом регистре. Далее, значение знаменателя отослать в память МК, а затем после вычисления числителя вызвать значение знаменателя из регистра памяти и произвести операцию деления.

Решение задачи на МК выглядит следующим образом:

$$\begin{aligned} l &\times f \text{ F ЗП } e \times k - \text{F ИП} = \text{F ЗП} \\ b &\times c \times d + a \div \text{F ИП} = \end{aligned}$$

Пример. Пусть в треугольнике  $ABC$  заданы значения стороны  $BC = a$  и углов  $B$ ,  $C$ . Необходимо вычислить значения  $b$ ,  $c$  и  $A$ , а также площадь треугольника  $S$ .

Известно, что  $A = \pi - (B + C)$ , если  $B$  и  $C$  заданы в радианах, и  $A = 180^\circ - (B + C)$ , если  $B$  и  $C$  заданы в градусах

$$b = \frac{a \sin B}{\sin A}, \quad c = \frac{a \sin C}{\sin A}, \quad S = \frac{1}{2} ac \sin B.$$

Для вычисления по этим формулам необходимо уметь находить значение  $\sin x$  по заданному аргументу.

Можно, как мы уже убедились, вычислить значение  $\sin x$  на МК, используя рассмотренный выше алгоритм. Однако в нашем распоряжении теперь имеется МК со специальными клавишами для вычисления элементарных функций, в том числе и  $\sin x$ .

Будем считать, что значения  $B$  и  $C$  заданы в градусах. Поставим переключатель ГРАД—РАД в положение ГРАД. Вычислим сначала значение угла  $A$ , а затем значение стороны  $b$ , имея в виду, что после предыдущей цепочки действий в регистре  $X$  находится значение угла  $A$ , которое используется нами на следующем шаге вычислений:

$$F \sin F \text{ ЗП } B \text{ F } \sin \times a \div F \text{ ИП} =$$

На индикаторе получим значение  $b$ , которое нужно зафиксировать на бумаге или запомнить.

Вычислим теперь значение стороны  $c$ . При этом воспользуемся вычисленным при подсчете  $b$  значением  $\sin A$ , которое находится в регистре памяти МК:

$$C \text{ F } \sin \times a \div F \text{ ИП} =$$

Для решения поставленной задачи осталось вычислить площадь треугольника  $S$ , используя полученные ранее значения сторон  $b$  и  $c$ . Отметим, что значение  $c$ , как результат последней операции, находится в регистре  $X$ :

$$\times B \text{ F } \sin \times a \div 2 =$$

На индикаторе будем иметь значение  $S$ .

Обратим еще раз внимание на тот факт, что при вычислении элементарных функций, в большинстве моделей МК используется только регистр  $X$ , содержимое же регистра  $Y$  при этом не меняется. Это обстоятельство мы использовали в рассмотренном выше примере, что позволило нам сократить число выполненных операций, необходимых для его решения.

**Пример.** Найти длину стороны  $c$  треугольника, если заданы стороны  $a$ ,  $b$  и угол между ними равен  $C$ .

Известно, что  $c = \sqrt{a^2 + b^2 - 2ab \cos C}$ . Для нахождения  $c$  достаточно проделать следующее:

вычисление  $a^2$

$$\begin{aligned} a \times &= \text{П} + b \times = \text{П} + 2 \times a \\ \times b \times C \text{ F } \cos &= \text{П} - \text{F ИП } F \text{ V} \end{aligned}$$

В этом примере обратим внимание, во-первых, на использование накапливающих регистров, во-вторых, на то, что при вычислении значения  $a^2$  мы вместо того, чтобы поступить так:  $a \times a =$ , сделали следующее:  $a \times =$ . Результат, однако, мы получим одинаковый в обоих случаях. В этом легко убедиться, если проследить за содержимым регистров МК:

Ввод	X	Y
$a$	$a$	
$\times$	$a$	$a$
$a$	$a$	$a$
$=$	$a^2$	$a$

Ввод	X	Y
$a$	$a$	
$\times$	$a$	$a$
$=$	$a^2$	$a$

Очевидно, что во втором случае мы сделали меньшее число манипуляций клавишами.

Таким приемом удобно пользоваться при вычислении степеней с произвольным натуральным показателем. Например, чтобы возвести число  $a$  в степень  $k$ , достаточно ввести это число  $a$  в МК и  $k-1$  повторить операцию умножения, а затем нажать клавишу  $=$ ; т. е.

$$a \underbrace{\times \dots \times}_{k-1 \text{ раз}} =$$

В результате получим  $a^k$ .

Рассмотрим еще один пример. Известно, что корни приведенного квадратного уравнения  $x^2 + px + q = 0$  в случае, если  $p^2/4 - q \geq 0$  вычисляются по формулам:

$$x_1 = -p/2 + \sqrt{p^2/4 - q}; \quad x_2 = -p/2 - \sqrt{p^2/4 - q}.$$

Если  $p^2/4 - q < 0$ , то уравнение не имеет действительных корней.

Алгоритм нахождения значений  $x_1, x_2$  на МК, где имеется регистр памяти и реализована функция  $\sqrt{a}$  ( $a \geq 0$ ), выглядит следующим образом:

$$p \times \div 4 - q =$$

На индикаторе МК получили значение  $p^2/4 - q$ . Если  $p^2/4 - q < 0$ , то вычисления прекращаем. Если  $p^2/4 - q \geq 0$ , то достаточно выполнить

$$F \vee F \text{ ЗП } p \div 2 \div 2 + F \text{ ИП} =$$

на индикаторе получили значение корня  $x_1$ .

Для вычисления  $x_2$  достаточно проделать следующее:

$$p \div 2 - F \text{ ИП} =$$

В настоящей главе изложены общие сведения о простейших МК. Здесь мы не стремились к тому, чтобы ознакомить читателя с правилами пользования каким-либо конкретным МК. Для этих целей каждое такое вычислительное устройство имеет соответствующую инструкцию. Цель изложенного выше — дать первоначальные сведения об МК, о его вычислительных возможностях. Основное внимание при знакомстве с МК было уделено приемам решения различных задач на простейших моделях МК. Очевидно, что если в вашем распоряжении имеется вычислительное устройство с большими функциональными возможностями, то это в значительной мере облегчит процесс вычислений и существенным образом расширит круг задач, которые можно эффективно решать с использованием МК.

Следует также подчеркнуть, что простейшие МК представляют своеобразный исполнитель алгоритмов и, как мы убедились на многих примерах, эффективность их использования зависит от того, насколько полно выбранный алгоритм решения задачи учитывает возможности используемого при ее решении МК. Простейшие преобразования исходных формул позволяют в ряде случаев найти алгоритм с меньшим числом шагов, сократить количество нажатий на клавиши. Работа на МК требует тем самым творческого подхода, овладения некоторыми навыками и опытом.

### § ПЗ. Программируемые микрокалькуляторы

При проведении расчетов часто возникает необходимость в выполнении вычислений по одному и тому же алгоритму для различных начальных данных. Например, нужно посчитать значение полинома  $P_n(x)$  при различных значениях  $x$ , найти корни квадратного уравнения для различных значений коэффициентов, решить треугольник для различных значений сторон и углов и т. д. Конечно, для этих целей можно использовать простейшие МК, с основами вычисления на которых мы познакомились ранее. Однако такое вычислительное устройство предполагает участие человека в каждом шаге вычислительного процесса (нужно задавать МК операнды операции и саму операцию). Причем для каждого набора исходных данных для получения результатов нужно всякий раз проделявать одну и ту же последовательность действий. Удобнее подобного рода задачи решать с помощью программируемых микрокалькуляторов (ПМК). Человеку в этом случае необходимо представить алгоритм решения задачи в виде набора инструкций на языке, понятном ПМК, т. е. написать для него программу. Далее ПМК может самостоятельно выполнить эту программу.

ПМК представляет собой довольно сложное вычислительное устройство с широкими функциональными возможностями. Им можно пользоваться аналогично тому, как это описано выше, т. е. вводить последовательно данные, указывать операцию и на каждом шаге полу-

чать результаты этой операции. Другими словами, принимать непосредственное участие в реализации каждого шага алгоритма решения задачи. Для этого предусмотрен специальный режим работы ПМК, в который он переводится при помощи клавиши АВТ (автоматическая работа). Основное назначение ПМК — производить вычисления автоматически по программе, заранее в него введенной. Ввод и изменение программы осуществляется в режиме программирования,

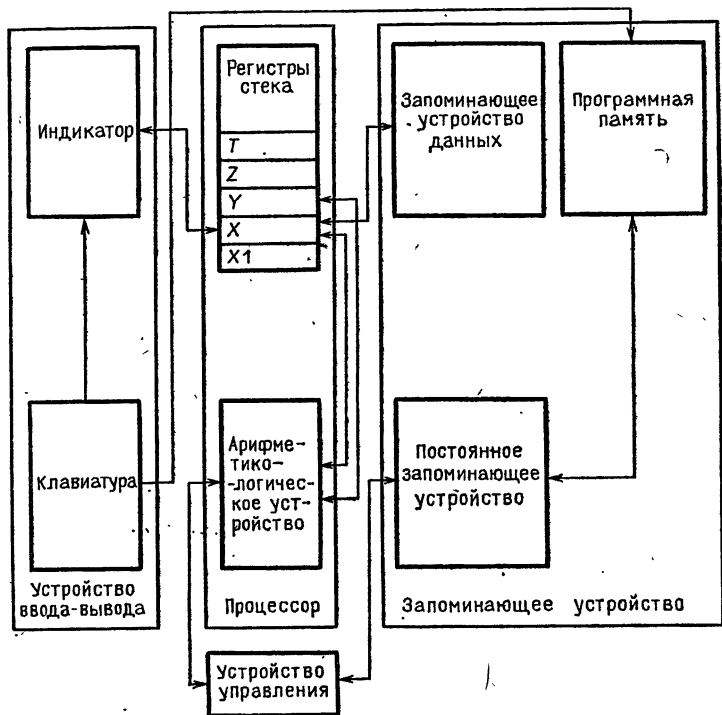


Рис. П4

для перевода ПМК в который служит специальная клавиша ПРГ. Упрощенная схема ПМК, отображающая взаимодействие его основных устройств, приведена на рис. П4.

Легко видеть, что ПМК содержит все основные устройства, которые входят в состав ЭВМ. А именно: устройство ввода—вывода, процессор, запоминающее устройство и устройство управления.

Устройство ввода—вывода ПМК состоит из клавиатуры и индикатора. Программа и числа, вводимые в ПМК при помощи клавиатуры, отображаются на индикаторе, что позволяет контролировать правильность выполняемых действий набора. Кроме чисел на инди-

каторе в процессе работы ПМК появляются и некоторые специальные символы, которые характеризуют режим его работы.

В дальнейшем, в процессе изложения материала, в основном будем ориентироваться на ПМК «Электроника МК-54», внешний вид которого представлен на рис. П5.

Клавиатура его состоит из 30 клавиш, причем две из них F и K имеют только одно функциональное назначение. 23 клавиши (они обведены на рис. П5 штриховой линией) — имеют по два функциональных назначения. Пять клавиш нижнего ряда имеют по три функциональных назначения. Так же как и в большинстве моделей МК для использования функционального назначения, обозначенного над клавишей, нужно предварительно нажать клавишу F. Значение НОП клавиши 0 приводится в действие предварительным нажатием клавиши K. Значения  $a$ ,  $b$ ,  $c$  и  $d$  клавиш нижнего ряда обеспечиваются предварительным нажатием либо клавиши  $P \rightarrow X$ , либо  $X \rightarrow P$ . Отметим, что действие, так называемых, префиксных клавиш F, K,  $P \rightarrow X$ ,  $X \rightarrow P$  обеспечивает соответствующую функцию клавиши, которая будет нажата непосредственно за каждой из них.

Запоминающее устройство ПМК разбито на несколько секций. В частности, имеется память для хранения программы, запоминающее устройство данных, постоянное запоминающее устройство.

Программная память представляет собой набор ячеек, куда заносятся коды команд. В одну ячейку можно записать только один код. Очевидно, что длина программы, которую можно ввести в ПМК, определяется количеством ячеек программной памяти. Число таких ячеек, например, в ПМК «Электроника МК-54» равно 98. Они пронумерованы двузначными числами от 00 до 97. ПМК автоматически фиксирует адрес текущей ячейки программной памяти, которая готова к приему команды. При вводе очередной команды в память ПМК этот адрес автоматически увеличивается на единицу. Следует отметить, что существует возможность изменения значения текущего адреса ПМК на заданное число как в сторону увеличения номера, так и в сторону его уменьшения. При вводе программы следует иметь в виду тот факт,

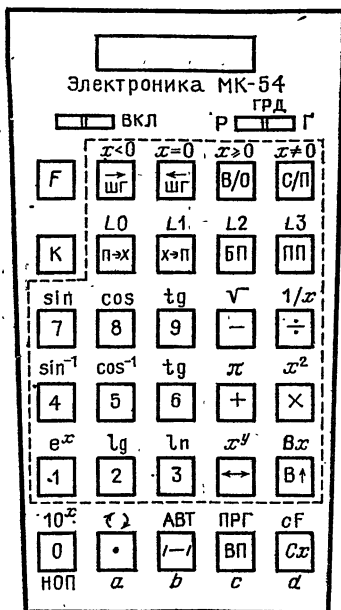


Рис. П5



что после заполнения всех ячеек памяти попытка ввести какую-либо новую команду приведет к стиранию информации, находящейся в ячейке с адресом 00, а затем и в следующих ячейках.

Запоминающее устройство данных позволяет одновременно хранить до 14 различных чисел; являющихся как исходными данными, так и результатами промежуточных вычислений. Оно реализовано в виде регистров, которые обозначаются буквами *a*, *b*, *c*, *d* и цифрами от 0 до 9.

Для записи числа в запоминающее устройство данных нужно набрать на клавиатуре это число последовательно цифра за цифрой, нажать клавишу  $X \rightarrow P$ , вслед за которой указать номер регистра. Например, в результате последовательного нажатия клавиш 1 6 3  $X \rightarrow P$  6 число 163 будет помещено в регистр с номером 6 запоминающего устройства данных.

Для считывания числа из какого-либо регистра запоминающего устройства данных необходимо нажать клавишу  $P \rightarrow X$  и вслед за ней указать номер регистра памяти, откуда следует считать число. Например,  $P \rightarrow X$  *a*. В результате чего копия содержимого регистра *a* попадет в регистр *X* и высветится на индикаторе ПМК.

Постоянное запоминающее устройство содержит большое число различных программ для выполнения арифметических и логических действий, а также для вычисления элементарных и специальных математических функций. Следует подчеркнуть, что мы можем лишь обращаться к этим программам и получать результат их работы, а изменять сами программы каким-либо образом не представляется возможным.

Процессор включает в свой состав арифметико-логическое устройство и специальные регистры (на схеме они обозначены *X1*, *X*, *Y*, *Z*, *T*). Регистры *X* и *Y* выполняют в ПМК те же функции, что и аналогичные регистры рассмотренных ранее моделей микрокалькуляторов. А именно: регистр *X* играет роль регистра ввода. Его содержимое высвечивается на индикаторе и выступает в качестве операнда выполняемой операции или аргумента вычисляемой функции. Регистр *Y* — операционный регистр. В нем находится другой операнд операции.

Регистры *X*, *Y*, *Z*, *T* образуют так называемый стек. Процессор выполняет все операции над операндами, которые берутся непосредственно из стека. Ниже мы остановимся на особенностях изменения чисел в стеке, отметим в данный момент лишь тот факт, что числа могут перемещаться по регистрам стека автоматически, после выполнения некоторых операций или в результате выполнения специальных команд.

Устройство управления регулирует работу всех узлов ПМК в различных режимах, задаваемых пользователем.

Для того чтобы работать на ПМК с программой, необходимо придерживаться следующей последовательности действий. Установить

режим программирования (ПРГ), с помощью клавиатуры ввести программу в программную память ПМК, после чего перейти в режим автоматических вычислений (АВТ) и ввести начальные данные в адресуемые регистры запоминающего устройства данных. Установить адрес, который определяет начало выполняемой программы и запустить программу на счет.

Полный цикл работы ПМК выглядит следующим образом:

Устройство управления считывает первую команду набранной вами программы, анализирует ее, вызывает из постоянного запоминающего устройства микропрограмму, необходимую для реализации команды, и передает ее для исполнения процессору, включающему арифметико-логическое устройство и устройство управления. После выполнения этой команды для исполнения выбирается следующая по порядку или отстоящая от нее на заданное число шагов команда набранной программы и т. д. до специальной команды останова.

Как уже отмечалось, ПМК обладает пятью специальными регистрами X1, X, Y, Z, T, четыре из которых (X, Y, Z, T) образуют стек. Принцип организации такой памяти состоит в том, что записанные в стек числа могут быть считаны и использованы в качестве операндов операций в последовательности, обратной той, в которой они записаны.

При вводе нового числа или выполнении некоторых операций автоматически происходит обмен содержимым между соседними регистрами стека.

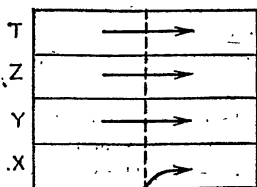
Ввод числа производится в регистр X. Для этого, как известно, достаточно последовательно набрать при помощи клавиатуры изображающие число знаки. Прежнее содержимое регистра X при этом уничтожается, а на его место запишется вводимое число. Содержимое остальных регистров останется без изменения.

Получившаяся при этом ситуация эквивалентна результату действия оператора присваивания

$X := a;$

где  $a$  — число, набранное с помощью клавиатуры.

Для большей наглядности изменение состояния регистров стека часто изображают схематично. В нашем случае мы получим следующую картину:

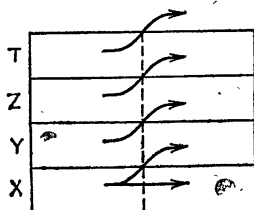


Если необходимо ввести подряд несколько чисел, то для ПМК требуется указание, где заканчивается ввод одного и начинается ввод другого числа. Для этого ПМК имеет специальную клавишу  $V\uparrow$ , после нажатия которой происходят следующие перемещения содержимого регистров стека:

$$T_n := Z_c; \quad Z_n := Y_c; \quad Y_n := X_c;$$

(здесь индекс  $n$  означает новое значение, а индекс  $c$  — старое значение содержимого соответствующего регистра).

В регистре  $X$  при этом сохраняется то значение, которое в нем было до выполнения операции, вызванной нажатием клавиши  $V\uparrow$ . Схематично эти перемещения выглядят так:

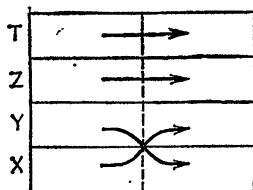


Заметим, что прежнее содержимое регистра  $T$  при выполнении операции  $V\uparrow$  теряется.

Обмен содержимым между регистрами  $X$  и  $Y$  осуществляется нажатием клавиши  $\leftrightarrow$ . При этом происходят следующие перемещения чисел в стеке

$$Y_n := X_c; \quad X_n := Y_c.$$

На схеме это выглядит так:

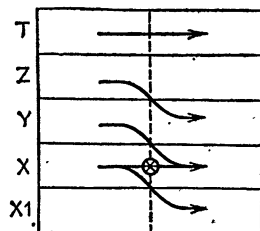


В ПМК, как и в обычном микрокалькуляторе, в качестве операндов при выполнении двуместных арифметических операций выступает содержимое регистров  $X$  и  $Y$ . Причем первым операндом операции является содержимое регистра  $Y$ , вторым операндом выступает содержимое регистра  $X$ . Результат операции помещается в регистр  $X$ .

Выполнение операции над операндами приводит к автоматическому перемещению чисел в стеке. Происходит это следующим образом:

$$X1_n := X_c; \quad X_n := Y_c \odot X_c; \\ Y_n := Z_c; \quad Z_n := T_c;$$

где  $\otimes$  — знак двуместной операции:

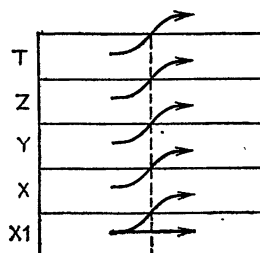


Следует отметить, что здесь перемещения данных затронули и регистр X1. Этот регистр часто называют регистром предыдущего результата, так как в нем сохраняется число, которое до выполнения операции находилось в регистре X.

Если теперь нам потребуется вызвать в регистр X число из X1, то для этого достаточно нажать клавиши F B↑. При этом произойдет следующее перемещение данных в регистрах стека

$$T_H := Z_C; Z_H := Y_C; Y_H := X_C; X_H := X1_C.$$

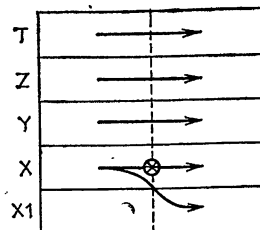
Схематично эти перемещения выглядят так:



При выполнении одноместных операций в качестве операнда выступает содержимое регистра X. Содержимое Y, Z, T остается неизменным, а копия числа из X передается в X1. Результат операции помещается в регистр X, т. е. происходит следующее:

$$X1_H := X_C; X_H := X_C \otimes$$

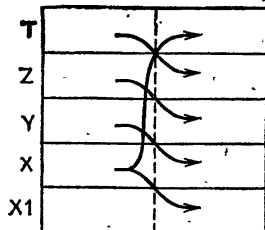
где  $\otimes$  — знак одноместной операции. На схеме эти перемещения выглядят так:



Анализируя перемещения информации в стеке, мы могли заметить, что иногда содержимое регистра Т теряется, что естественно может оказаться нежелательным. Часто также возникает необходимость просмотреть содержимое всех регистров стека либо определенным образом осуществить перемещение чисел в стеке. Для этой цели предназначена специальная операция, которая задается последовательным нажатием клавиш  $F \curvearrowright$ . При этом произойдет следующее перемещение чисел в стеке:

$$\begin{aligned} X1_n &:= X_c; & X_n &:= Y_c; \\ Y_n &:= Z_c; & Z_n &:= T_c; & T_n &:= X1_n. \end{aligned}$$

На схеме эти перемещения выглядят так:



Очевидно, что для вызова в регистр X, а, следовательно, и на экран индикатора содержимого регистра Т, необходимо три раза воспользоваться клавишей  $F \curvearrowright$ .

Для очистки всех регистров стека достаточно нажать клавишу  $Sx$ , в результате чего в регистр X занесется ноль. После этого нужно трижды нажать клавишу  $V \uparrow$ . Это приведет к тому, что в регистрах Y, Z и T окажутся нули.

Мы познакомились с некоторыми конструктивными особенностями ПМК, в частности, наличием стека и правилами перемещения в нем чисел. Естественно, что эти особенности в значительной мере определяют организацию процесса вычислений на ПМК. Знание этих особенностей позволит оптимальным образом производить вычисления на ПМК.

**Пример.** Пусть нам необходимо вычислить значение выражения вида

$$S = \sqrt{(a+b)(c-d)}.$$

Ход вычисления значения  $S$  можно описать следующим образом:

- взять число  $a$ ,
- взять число  $b$ ,
- сложить  $a$  и  $b$ ,
- взять число  $c$ ,
- взять число  $d$ ,
- вычесть  $d$  из  $c$ ,
- умножить  $(a+b)$  на  $(c-d)$ ,
- извлечь квадратный корень из  $(a+b)(c-d)$ .

Сокращенно указание для вычисления  $S$  можно записать, указав операнды и вслед за ними знаки операций:

$$a b + c d - \times \sqrt{\phantom{x}}$$

Такая запись носит название польской инверсной записи, она не использует скобок, в ней не нужно заботиться о старшинстве операций. Знак операции помещается непосредственно после операндов и относится к последнему или двум последним числам (или результатам).

Правила вычисления по формуле, представленной в таком виде, состоят в следующем. Идя слева направо, выбираются два операнда (или один, если операция одноместная), над ними производится операция, обозначение которой следует вслед за операндами. Результат этой операции в дальнейшем рассматривается как операнд новой операции.

В нашем случае на первом шаге выполняется операция сложения, где операндами выступают  $a$  и  $b$ . Далее берется следующая пара чисел  $c$  и  $d$ , которые рассматриваются как операнды операции вычитания.

На следующем шаге произойдет перемножение результатов первого и второго шагов, т. е. получим  $(a+b) \times (c-d)$ . И, наконец, этот результат явится операндом последней операции — извлечения квадратного корня из  $(a+b)(c-d)$ .

Как оказалось, такая форма записи удобна для исполнения на многих ЭВМ, включая ПМК. Вспомним, что в этой записи операция выполняется над расположенными подряд операндами, а ПМК обладает стеком — набором регистров, между которыми при вводе чисел и выполнении операций происходит согласованный обмен содержимым.

Проследим за содержимым регистров стека ПМК при вычислении значения  $S$  при  $a=10$ ,  $b=2$ ,  $c=8$ ,  $d=1$ :

Операция	X	Y	Z	T	X1
10	10	0	0	0	0
↑	10	10	0	0	0
2	2	10	0	0	0
+	12	0	0	0	2
8	8	12	0	0	0
↑	8	8	12	0	0
1	1	8	12	0	0
—	7	12	0	0	1
×	84	0	0	0	7
F	84	0	0	0	7
√	9,1651513	0	0	0	84

Перепишем в строчку последовательность нажатия клавиш

1 0 В↑ 2 + 8 В↑ 1 — × F √

Вообще говоря, указанная последовательность нажатия клавиш и представляет собой программу решения задачи на микрокалькуляторе. Задачу можно решать вручную, т. е. последовательно нажимая необходимые клавиши в режиме АВТ, или автоматически, предварительно записав программу в программную память ПМК в режиме ПРГ.

#### § П4. Основные приемы программирования ПМК

Для проведения вычислений по программе в первую очередь нужно написать программу решения задачи, т. е. последовательность команд ПМК, необходимых для ее решения, затем ввести эту программу в программную память ПМК, задать исходные данные и наконец дать ПМК команду на выполнение программы.

Рассмотрим простейшую задачу. Вычислить

$$S = \ln(x + \sqrt{x^2 + 1}).$$

Перепишем выражение, стоящее справа от знака равенства, в польской инверсной записи

$$x \ x \times 1 + \sqrt{x + \ln}$$

Легко видеть, что для решения поставленной задачи достаточно произвести следующие действия:

$$x \ В↑ \times 1 + F \sqrt{+ F \ln}$$

Здесь команда В↑ осуществляет пересылку значения  $x$  из регистра индикатора  $X$  в регистр  $Y$ . После выполнения команды  $\times$  в регистре  $X$  будет находиться  $x^2$ , а в регистре  $Y$  — значение  $x$ . В результате занесения в регистр  $X$  единицы значение  $x^2$  из регистра  $X$  переместится в регистр  $Y$ , откуда значение  $x$  будет предварительно перенесено в регистр  $Z$ . Операция  $+$  даст значение  $x^2 + 1$ , которое будет находиться в регистре  $X$ .

Регистр  $Y$  при этом будет содержать значение  $x$ . После извлечения квадратного корня из содержимого регистра  $X$ , выполнится команда  $+$ , в результате чего в регистре  $X$  окажется  $(x + \sqrt{x^2 + 1})$ . И, наконец,  $F \ln$  обеспечит вычисление значения натурального логарифма числа, находящегося в регистре  $X$ . На индикаторе получим значение  $\ln(x + \sqrt{x^2 + 1})$ .

Таким образом, рассмотренная выше последовательность действий представляет собой программу решения задачи для произвольных значений  $x$ .

Сделаем два замечания.

1. Так как предполагается автоматическое выполнение программы, то необходимо предусмотреть останов работы ПМК после выполнения

последней команды программы. После выполнения команды останова прекращается выполнение программы и текущее содержимое регистра X фиксируется на индикаторе. Для задания команды останова служит клавиша С/П.

2. Выполнение программы осуществляется, как правило, для некоторого набора исходных данных. Например, в нашем случае вычисления можно осуществлять для произвольных значений  $x$ . Исходные данные следует задавать программе перед пуском ее на счет. Сделать это можно, например, непосредственно ввода в стек ПМК необходимых числовых значения, либо засылая их в ячейку памяти данных. В последнем случае в программе нужно предусмотреть возможность вызова этих данных из памяти.

В нашем примере, если значение  $x$ , для которого вычисляется  $S$ , находится в регистре X, программа примет вид

$$B\uparrow \times 1 + F \sqrt{\phantom{x}} + F \ln C/P$$

Если же  $x$  находится в каком-либо регистре данных, например, в регистре с номером 1, то перед  $B\uparrow$  следует задать  $P \rightarrow X 1$  — команду вызова числа из регистра памяти с номером 1 в регистр X.

Для удобства работы с программой, как правило, указывают адреса команд программы в программной памяти ПМК, а иногда и коды операций, которые соответствуют нажимаемым клавишам. (Соответствие кодов и команд для каждого ПМК приводится в инструкции по его эксплуатации.)

В этом случае наша программа примет следующий вид:

00	01	02	03	04	05	06	07
0E	12	01	10	21	10	18	50
$B\uparrow \times$		1 +	$F \sqrt{\phantom{x}}$		+	$F \ln$	C/P

Здесь верхняя строка — адреса команд, следующая за ней — коды команд, нижняя строка — клавиши на пульте ПМК.

Отметим, что программа решения задачи может начинаться с произвольного адреса. Однако чаще всего в качестве начального выбирают адрес 00.

Для очистки программного счетчика необходимо в режиме «Автоматическая работа» нажать клавишу В/0 и перейти в режим «Программирование», нажав клавиши F ПРГ. На индикаторе при этом в правом углу высветятся цифры 00, обозначающие адрес, с которого вводится программа. Если нужно вводить программу в ПМК начиная с некоторого фиксированного адреса, отличного от 00, то для этого в режиме «Автоматическая работа» достаточно нажать клавишу БП и вслед за ней при помощи клавиатуры задать двузначное число, равное этому адресу. Например, БП 0 9 означает, что первая команда программы будет помещена в ячейку с адресом 09.



Ввод программы в ПМК осуществляется с помощью клавиатуры последовательно команда за командой. В процессе ввода для контроля правильности набора на индикаторе высвечиваются четыре пары символов

$$D_1D_2 \ C_1C_2 \ B_1B_2 \ A_1A_2,$$

где  $A_1A_2$  — адрес команды в памяти, которая будет вводиться на следующем шаге;  $D_1D_2$  — код последней введенной в ПМК операции, которая находится по адресу на единицу меньше, чем значение  $A_1A_2$ .  $C_1C_2$  и  $B_1B_2$  соответственно коды предпоследней введенной команды и команды введенной непосредственно перед ней. Адреса этих команд в программной памяти соответственно на два и на три меньше адреса  $A_1A_2$ .

Например, в нашем случае при занесении программы на индикаторе последовательно будем иметь:

F	ПРГ		00
B	†	0E	01
X		12 0E	02
1		01 12 0E	03
+		10 01 12	04

В процессе ввода программы в ПМК при наборе той или иной команды может быть допущена ошибка. Иногда возникает необходимость во внесении исправлений в программу. Для просмотра программы с целью проверки правильности ее записи в ПМК, а также для редактирования в случае обнаружения ошибок, либо модификации ее отдельных частей используются клавиши  $\overline{\text{ШГ}}$  и  $\overline{\text{ШГ}}$ . Первая из этих двух клавиш позволяет осуществлять просмотр программы, находящейся в памяти ПМК, «вперед». При каждом ее нажатии мы будем на экране получать код команды, адрес которой на единицу больше текущего.

Соответственно клавиша  $\overline{\text{ШГ}}$  обеспечивает просмотр программы «назад» последовательно команда за командой.

Для того чтобы исправить какую-либо команду из ранее введенных в ПМК, необходимо после появления на индикаторе адреса этой команды (две крайние правые цифры) набрать с помощью клавиатуры правильную команду.

Можно также вообще исключить любую команду из программы.

Для этого необходимо при помощи клавиш  $\overline{\text{ШГ}}$   $\overline{\text{ШГ}}$  получить на индикаторе адрес этой команды и нажать последовательно клавиши К НОП. Таким образом, используя рассмотренные выше средства, мы можем ввести программу в ПМК, проверить правильность ввода и при необходимости отредактировать программу.

Наша программа находится в программной памяти ПМК, мы убедились в том, что она записана туда верно. Теперь необходимо перейти в режим «Автоматическая работа», т. е. нажать клавиши F АВТ и занести исходные данные. В нашем случае достаточно набрать при помощи клавиатуры значение  $x$ .

Для того чтобы начать вычисления, нужно сообщить ПМК адрес первой команды программы.

В нашем случае он равен 00, поэтому необходимо нажать клавишу В/0. В случае, если программа начинается с некоторого адреса отличного от нулевого, то, как уже говорилось, нужно нажать клавишу БП и задать начальный адрес программы.

Пусть теперь программа находится в памяти ПМК, заданы исходные данные, ПМК сообщен адрес начала программы и ее нужно запустить на счет. Для этих целей служит клавиша С/П. После выполнения программы на индикаторе мы получим результат.

Для контроля правильности программы, удобно пользоваться клавишей ПП. Она позволяет в режиме «Автоматическая работа» выполнять программу последовательно по отдельной команде и на индикаторе на каждом шаге получать результат выполнения команды. Это дает возможность осуществлять отладку программы, обнаруживать ошибки не только в программе, но и в алгоритме решения задачи.

Выпишем теперь последовательность действий для проведения вычислений по программе на ПМК.

1. В/0 (установка нулевого адреса для ввода программы).
2. F ПРГ (перевод ПМК в режим «Программирование»).
3. Ввод программы.
4. F АВТ (перевод в режим «Автоматическая работа»).
5. Ввод начальных данных.
6. В/0 (установка адреса начала программы).
7. С/П (пуск программы на счет).
8. Запись и анализ результатов.

Если теперь провести вычисления значения  $S$  по нашей программе для  $x=1$ , то получим на индикаторе число 8.813755—01.

Для того чтобы вычислить  $S$  при другом значении  $x$ , достаточно выполнить последовательность действий, начиная с пункта 5, т. е. набрать с помощью клавиатуры новое значение  $x$ , установить адрес начала программы (В/0), и запустить программу на счет (С/П).

С помощью ПМК можно успешно решать такие задачи, в которых предполагается переход на ту или иную ветвь алгоритма в зависимости от результата проверки какого-либо условия.

Например, вычислим

$$S = \begin{cases} (x+3)/2, & \text{если } 2x+1 \geq 0, \\ x/3-2, & \text{если } 2x+1 < 0. \end{cases}$$

Составим блок-схему алгоритма вычисления  $S$  (рис. П6).

Очевидно, что для решения задачи необходимо предусмотреть возможность вычисления значений трех арифметических выражений:

- 1)  $U = 2x + 1$ ,
- 2)  $S = (x + 3)/2$ ,
- 3)  $S = x/3 - 2$ .

Перепишем эти выражения в польской инверсной записи

- 1)  $2 \ x \times 1 \ +$
- 2)  $x \ 3 \ + \ 2 \ /$
- 3)  $x \ 3 \ / \ 2 \ -$

(здесь переменной  $x$  поставлен в соответствие четвертый регистр) и составим программу вычисления каждого из выражений 1—3. (При записи программ здесь и в дальнейшем будем писать только адреса команд в памяти ПМК и соответствующие им клавиши. Значения кодов команд для краткости опустим.)

1.  $P \rightarrow X \begin{matrix} 00 & 01 & 02 & 03 & 04 \\ 4 & 2 & \times & 1 & + \end{matrix}$
2.  $P \rightarrow X \begin{matrix} 10 & 11 & 12 & 13 & 14 & 15 \\ 4 & 3 & + & 2 & \div & C/P \end{matrix}$
3.  $P \rightarrow X \begin{matrix} 20 & 21 & 22 & 23 & 24 & 25 \\ 4 & 3 & \div & 2 & - & C/P \end{matrix}$

Мы получили три программы, каждая из которых вычисляет соответствующее арифметическое выражение.

Числовой результат работы первой программы нас не интересует, важен лишь знак результата. Сразу после ее выполнения вычисление

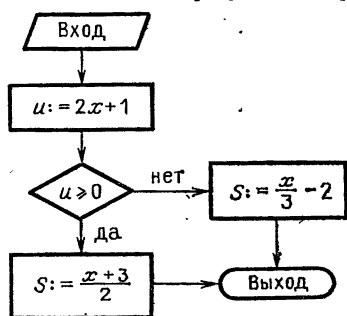


Рис. П6

должно продолжаться либо по программе 2, либо 3, что даст нам ответ задачи. В связи с этим программы 2 и 3 завершаются командой С/П для останова ПМК и фиксации на его индикаторе результата. В памяти ПМК программы 1—3 будут располагаться так:

первая — в ячейках с номерами 00—04, вторая — 10—15, третья — 20—25. В зависимости от результата вычисления по программе 1,

мы должны уметь осуществлять переход либо к программе 2, либо 3. Для этих целей в ПМК предусмотрены так называемые команды перехода по условию ( $X \geq 0$ ,  $X < 0$ ,  $X = 0$ ,  $X \neq 0$ ), которые реализуются последовательным нажатием клавиши F, одной из клавиш  $X \geq 0$ ,  $X < 0$ ,  $X = 0$ ,  $X \neq 0$ , а также клавиш, задающих двузначное число — адрес перехода. Например, F X=0 0 6.

Смысл команд перехода по условию состоит в проверке содержимого регистра X на выполнение требуемого условия. Если условие не соблюдается, то следующей будет выполнена команда, находящаяся в программной памяти по адресу, значение которого совпадает с числом, являющимся адресом перехода в команде перехода по условию. Если же условие соблюдено, то выполняется следующая по порядку команда (т. е. перехода не происходит).

Наряду с командами условного перехода в ПМК предусмотрена и команда безусловного перехода. Задается эта команда нажатием уже упоминавшейся нами ранее клавиши БП, вслед за которой набирается двузначное число — адрес перехода. Выполнение команды безусловного перехода прерывает естественный порядок выполнения программы и вызывает для исполнения команду, адрес которой указан в адресе перехода. Например, БП 6 3.

Обратим внимание на следующее. Структура команд перехода по условию и безусловного перехода предполагает наличие в команде адреса перехода — двузначного числа. В связи с этим при записи этих команд в ПМК используются две подряд идущие ячейки памяти. В первую из них помещается код команды, во вторую адрес перехода. Например,

$$\dots \overset{15}{F} \overset{16}{X=0} \overset{17}{0} \overset{18}{8} \dots \overset{21}{БП} \overset{22}{1} \overset{23}{5} \dots$$

Здесь команда перехода по условию размещается вместе с адресом перехода в ячейках программной памяти с номерами 15 и 16. Она сравнивает содержимое регистра X ПМК с нулем. Если текущее значение регистра X равно нулю, то следующая для исполнения команда будет взята из ячейки памяти с номером 17. Если же содержимое регистра отлично от нуля, то управление будет передано команде из ячейки с номером 08. Команда безусловного перехода, находящаяся в нашем примере в ячейках 21—22, определит в качестве следующей для исполнения команду, адрес которой равен 15.

Вновь вернемся к задаче вычисления значения S. Объединим программы 1—3. Для этого добавим в программу 1 команду  $F X \geq 0 \ 2 \ 0$ . Если значение выражения  $2x+1$ , вычисленного перед выполнением команды перехода по условию и находящегося в регистре X, меньше нуля, то вычисления будут производиться по программе 3, первая команда которой находится в памяти по адресу—20. Если же  $2x+1 \geq 0$ , то согласно алгоритму необходимо произвести вычисления по программе 2, первая команда которой находится в ячейке с номером 10. Однако команда перехода по условию в этом случае передает управление команде из ячейки с номером 7, так как сама команда перехода займет ячейки 5 и 6.

Для того чтобы попасть на начало программы 2, достаточно вслед за командой перехода по условию поместить команду безусловного перехода БП 1 0, которая разместится в ячейках 7 и 8.

В этом случае программа примет вид

$$\begin{array}{l} \text{П} \rightarrow \text{X} \begin{array}{cccccc} 00 & 01 & 02 & 03 & 04 & 05 \\ 4 & 2 & \times & 1 & + & \text{F} \end{array} \text{X} \geq 0 \begin{array}{ccc} 06 & 07 & 08 \\ 2 & 0 & \text{БП} \end{array} 1 \quad 0 \\ \text{П} \rightarrow \text{X} \begin{array}{cccccc} 10 & 11 & 12 & 13 & 14 & 15 \\ 4 & 3 & + & 2 & \div & \text{С/П} \end{array} \\ \text{П} \rightarrow \text{X} \begin{array}{cccccc} 20 & 21 & 22 & 23 & 24 & 25 \\ 4 & 3 & \div & 2 & - & \text{С/П} \end{array} \end{array}$$

Вообще говоря, мы получили программу, которую можно вводить в память ПМК и проводить по ней вычисления.

Однако, если мы обратимся к строкам записи программы, где указаны адреса команд, то заметим, что собственно программа, выполняющая вычисления и выбор перехода по условию занимает ячейки памяти с номерами 00—08, 10—15, 20—25. Некоторое количество ячеек памяти из интервала 00—25 не используется.

Для обхода свободных участков памяти (куда, кстати, могут помещаться команды программы для решения другой задачи) приходится использовать команды безусловного перехода, которые требуют как место в памяти для хранения, так и время для выполнения. На практике, как правило, стремятся к тому, чтобы программа не имела подобных «дыр» при ее расположении в памяти ПМК.

С учетом последнего замечания вновь перепишем программу.

$$\begin{array}{l} \text{П} \rightarrow \text{X} \begin{array}{cccccc} 00 & 01 & 02 & 03 & 04 & 05 \\ 4 & 2 & \times & 1 & + & \text{F} \end{array} \text{X} \geq 0 \begin{array}{ccc} 06 & & \\ & & 13 \end{array} \\ \text{П} \rightarrow \text{X} \begin{array}{cccccc} 07 & 08 & 09 & 10 & 11 & 12 \\ 4 & 3 & + & 2 & \div & \text{С/П} \end{array} \text{П} \rightarrow \text{X} \begin{array}{ccc} 13 & 14 & \\ 4 & 3 & \\ \div & 2 & - \end{array} \text{С/П} \end{array}$$

В таком виде программу уже можно вводить в память ПМК.

Последовательность действий при вычислении по программе следующая:

1. В/0
2. F ПРГ
3. Ввод программы.
4. F АВТ
5. а X → П 4

(ввод в регистр числовой памяти 4 исходного значения  $x=a$ ).

6. В/0 С/П

7. Прочитать результат на индикаторе ПМК.

Например, при  $a=1$  получим  $S=2$ , а при  $a=-2$ ,  $S=-2,666667$ .

В ПМК предусмотрена возможность осуществлять циклические вычисления. Как известно, организовать циклы можно и с использованием команд условного и безусловного перехода. Однако в ПМК имеются специальные команды для организации циклов.

Структура таких команд имеет вид

LN *nn*,

где *N* является числом из набора {0, 1, 2, 3} и обозначает номер регистра, который используется для организации цикла; *nn* — двузначное число из интервала 00—97, являющееся адресом перехода.

Задают команды цикла с помощью клавиш (L0 L1 L2 L3), после чего необходимо набрать двузначное число — адрес перехода.

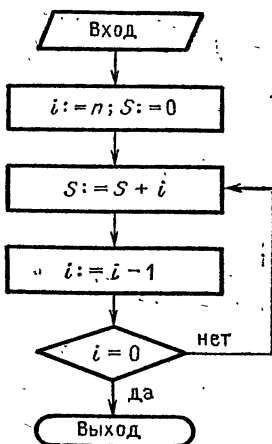


Рис. П7

При каждом нажатии клавиши L0 (L1, L2, L3) анализируется содержимое регистра с номером 0 (1, 2, 3). Если оно равно нулю, то выполняется команда, следующая непосредственно за командой цикла. В противном случае из содержимого регистра 0 (1, 2, 3) вычитается 1 и будет выполнена команда, ссылка на которую указана в адресе перехода команды цикла. Отметим, что при записи в память ПМК команда цикла занимает две подряд идущих ячейки.

Пример. Вычислить

$$S = \sum_{i=1}^N i = 1 + 2 + \dots + (N-1) + N.$$

Порядок проведения вычислений отражает блок-схема, представленная на рис. П7. Очевидно, что суммирование здесь происходит в обратном порядке.

Программа для ПМК вычисления *S* имеет следующий вид:

X → П 0, 0, П → X 0 + F L0 1 2 C/П

Для организации цикла в данном случае используется регистр с номером 0. Команда с адресом 10 помещает содержимое регистра X

в регистр памяти 0, что соответствует на блок-схеме оператору присваивания  $i := N$ . Тем самым мы задаем максимальное число из ряда суммируемых чисел.

Нуль, находящийся в ячейке по адресу 11, обнуляет содержимое регистра X перед началом процесса суммирования. Далее идут три команды, расположенные в ячейках 12—15, которые выполняются в цикле. Причем первая из них вызывает из регистра с номером 0 значение текущего слагаемого в сумме. Вторая производит операцию сложения очередного слагаемого с текущей суммой всех ему предшествовавших. Третья команда осуществляет формирование очередного слагаемого в сумме и управляет циклом.

Последовательность действий при вычислении по этой программе следующая:

1. БП 1 0
2. F ПРГ
3. Ввести программу.
4. F АВТ
5. Набрать на клавиатуре данные (число  $n$ ).
6. БП 1 0
7. С/П
8. Прочитать ответ на индикаторе.

Программирование для микрокалькуляторов демонстрирует еще один пример машинно-зависимого программирования. В случае набора программы для запоминания ее в программной памяти ПМК мы повторяем нажатие тех же функциональных клавиш, которые мы нажимали бы при «ручной» пошаговой работе. В какой-то степени можно говорить о языке программирования для ПМК, даже можно говорить о символьном машинно-зависимом языке, так как все клавиши помечены некоторыми символами. Правда, на индикаторе эти символы не высвечиваются, поэтому программист должен знать цифровые коды команд. В настоящее время разработаны ПМК, на индикаторе которых высвечиваются не только цифры, но и буквы, что позволяет проще контролировать правильность набора программ.

Как и для обычных ЭВМ программа для ПМК сначала составляется на бумаге. Ее стараются тщательно проверить до ввода в ПМК, затем программу вводят в программную память ПМК, осуществляют пошаговое выполнение на простых исходных данных, правят, если нужно, и только после этого, убедившись в том, что программа составлена правильно, переходят к вычислениям с нужными исходными данными. Очевидно, что все этапы программирования, которые необходимо выполнять для ЭВМ, присутствуют и в случае работы с ПМК, хотя каждый из этих этапов обладает своей спецификой. Творческий подход к овладению приемами работы на ПМК открывает его широкие возможности, позволяет рационально использовать ПМК для решения разнообразных задач.

## ТАБЛИЦА ОСНОВНЫХ ТЕХНИЧЕСКИХ ХАРАКТЕРИСТИК ПЭВМ СОВЕТСКОГО ПРОИЗВОДСТВА

### Пояснения к таблице.

Данные, помещенные в таблице, взяты из журнала «Микропроцессорные средства и системы», редакция которого получила их от проектировщиков и разработчиков соответствующих микроЭВМ.

При освоении микроЭВМ в серийном выпуске некоторые параметры могут отличаться от приведенных в таблице.

В первой графе таблицы приведены, как правило, названия микроЭВМ, принятые в литературе. Кроме того, как и любому другому изданию, каждому типу микроЭВМ присваивается шифр изделия. Так, например, микроЭВМ Электроника УКНЦ значится под шифром СМС 0202; ДВК-3 имеет шифр МС 0502; Электроника-85—МС 0585 и т. д.

Вторая графа характеризует разрядность микропроцессора, используемого в микроЭВМ, она определяет точность выполнения простейших арифметических и логических операций. Например, 8 разрядов соответствуют точности всего в две десятичных цифры. Грубо говоря, такой микропроцессор умеет считать только до 99. Чтобы заставить микроЭВМ на основе такого микропроцессора работать с большей точностью, для выполнения каждой арифметической операции требуется некоторая последовательность элементарных действий.

В третьей графе указано номинальное быстродействие при выполнении простейших арифметических операций. К этому показателю надо относиться с большой осторожностью. Так, например, для 8-разрядной ЭВМ быстродействие в 500 тыс. операций/с это не то же самое, что быстродействие в 500 тыс. операций/с для 16-разрядной ЭВМ.

При выполнении арифметических операций над числами в десятичной системе счисления, имеющими 10—12 десятичных знаков, это номинальное быстродействие падает в 20—30 раз, т. е. превращается в быстродействие, эквивалентное 10—20 тыс. операций/с



для 8-разрядной микроЭВМ и в 20—40 тыс. операций/с для 16-разрядной ЭВМ.

Четвертая графа указывает объем оперативной памяти, доступной пользователю. В действительности, каждая микроЭВМ обладает памятью большего объема, но не вся эта память находится в распоряжении пользователя.

В пятой графе указаны объемы внешней памяти. НГМД — означает накопитель на гибких магнитных дисках; НМД — накопитель на «жестких» дисках типа Винчестер.

В следующей графе указан тип дисплея. Когда речь идет о символьном (алфавитно-цифровом дисплее), то произведение  $24 \times 80$  обозначает, что на экране можно разместить 24 строки по 80 символов в каждой.

Для графических дисплеев произведение, например  $280 \times 440$ , означает общее число точек, размещаемых на экране, из которых можно формировать изображение, или, что то же самое, число строк в развертке и число точек в строке развертки.

В таблице приведены самые общие и далеко не полные сведения о математическом обеспечении. В основном это тип операционной системы и используемые языки. Программное обеспечение микроЭВМ очень быстро развивается, появляются новые языки, новые версии операционных систем, новые пакеты прикладных программ.

Седьмая графа в силу этого дает весьма приблизительные сведения о программном обеспечении.

939-15-04  
تلفون افراس  
931  
A.

Название или заводской шифр	Разряд- ность микро- процес- сора	Нормаль- ное быстро- действие, тыс. оп/с	Объем опе- ративной памяти пользо- вателя, Кбайт	Тип и объе- мы внешней памяти, Кбайт	Тип дисплея (параметры строк и столбцов)	Программное обеспечение	Примечания
ДВК-1	8	500	56	—	символьный 24×80	бейсик, ДСП, ОС ДВК	совместим по опе- рационной систе- ме с СМ 3/4
ДВК-2	8	500	56	НГМД 220—440	То же	То же	То же
ДВК-2М	8	500	56	НГМД 2×220	»	»	»
ДВК-3	16	800	64—248	НГМД 440—800	графический чер- но-белый 280×440	бейсик, паскаль, СИ, ОС ДВК	совместим с СМ-4
ДВК-3М2	16	800	64—248	НГМД 440—800	То же	То же	То же
ДВК-4	16	800	64—196	НГМД 800—1600	графический цвет- ной 280×440	бейсик, паскаль, фортран, ОС ДВК, ДСП	профессиональный
Электрон- ка 85	16	600	512	НГМД 2×400 НМД 5—10 Мбайт	символьный и гра- фический 240×960	бейсик, паскаль, фортран, ОС	

Название или заводской шифр	Разрядность микропроцессора	Нормальное быстродействие, тыс. оп/с	Объем оперативной памяти пользователя, Кбайт	Тип и емкость внешней памяти, Кбайт	Тип дисплея (параметры строк и столбцов)	Программное обеспечение	Примечания
ЕС 1840	16	500	256—640	НГМД 2×320	символьный черно-белый 25×80	ОС М86 (СР/М-86), бейсик, ППП АБАК, модуль-2,	профессиональный
ЕС 1841	16	500	256—640	НГМД НМД 5—10 Мбайт	графический цветной	То же	То же
ИСКРА	16	800	256—640	НГМД	графический	ОС АДОС, бейсик и др.	»
Электроника УКНЦ	16	800	64—192	НГМД 800	графический цветной 288×640	ОС ДВК, бейсик, паскаль и др.	»
Корвет	8—16	400	64—512	магнитофон 200—10 000 НГМД 512	символьный 16×80 графический 256×512	ОС Корвет, СР/М, ОС ДВК, бейсик, рапира, СИ	школьный, рабочее место учителя
БК 0010	8	300	16	—	символьный графический 256×512	фокал, бейсик	бытовой, школьный

## ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- Автокод 129, 166, 265
- Автоматизация- программирования 131
- Адрес 82
  - базовый 91
  - исполнительный 149
  - операнда 87
  - переменный 155
- Алгол 170
- Алгоритм 10, 18, 22, 32, 37, 78, 98, 115, 123, 167
  - линейный 118
  - оптимальный 31
  - разветвляющийся 124
- Алгоритма исполнитель 78
  - массовость 17
  - однозначность 18
  - определенность 18
  - неприменимость 23
  - применимость 23
  - эквивалентность 24
- Алгоритмизация 31, 32, 37
- Алгоритмическая неразрешимость 19
  - система Маркова 124
- Алфавит 171
- Анализ изображений 35
- Архитектура ЭВМ 7
- Ассемблер 129
- Атанасов А. 6
  
- Базисные числа 92
- Байт 82, 169, 225, 256, 264
- Бейбидж Чарльз 5
- Бейсик 165, 172
- Бейсик-система 217
- БИС 8
- Бит 82, 84
- Блок данных 179
- Блок-схема 124
  
- Быстродействие ЭВМ 7, 8, 9, 262
- Вектор 174
- Вложенность подпрограмм 163
- Возврат из подпрограммы 160
- Время доступа 83
- Выражение 176
- Вычислительные алгоритмы 40
  - комплексы 9
  - сети 9
- Вычислительный процесс 118
  
- Данные 10
  - исходные 17, 20, 21
  - недопустимые 23
- Джон фон Нейман 7
- Диалоговый режим 172, 207, 216, 267
- Диапазон представления чисел 133
- Директива 217
  - RUN 217
  - DELETE 218
  - LIST 218
- Дискрет 250
- Дисплей 85, 209, 248
  
- Заголовок цикла 191
- Запись алгоритма 115
  - словесно-формульная 124
- Запоминающее устройство 6, 79
  
- Идентификатор 174, 199, 225, 255
- Имя переменной 174
  - массива 174
- Индекс 174
- Интерполирование квадратичное 53

Интерполирование линейное 53  
— многочленами 50, 75  
Интерпретация 170  
Интерфейс 267  
Информатика 12  
Информации носитель 10  
Информация 10  
Исполнитель алгоритма 18, 20  
Исходные данные 21, 23

Каналы связи 79  
Кибернетика 12  
Клавиши ввода 273  
— операционные 273  
Клеточный автомат 38  
Кобол 170  
Код 81, 91, 169  
— операции 87, 130  
Кодирование 81, 92  
— символьное 127  
Команда 82, 112  
— арифметическая 161  
— возврата 161  
— вычитания 161  
— системная 217  
— сложения 161  
— пересылки 140  
— передачи управления 87  
— перехода 160  
— — безусловного 138  
— — на подпрограмму 217  
— умножения 161  
Комментарий 206  
Компиляция 170

Магнитные барабаны 82, 209  
— диски 82, 209  
— ленты 82, 209  
Мантисса числа 108, 113, 133, 273  
Массив 151, 174, 189, 199  
Математическая модель 10, 26, 29, 40  
Математическое обеспечение 216, 262, 267  
Матрица 175, 228  
Машинная графика 35, 247  
— операция 86  
машинное слово 82, 106, 108, 264  
Метод деления отрезка пополам 61, 122  
— касательных 63  
— наименьших квадратов 55

Метод хорд 65  
Микрокалькулятор 271  
— программируемый 271, 293  
Многопроцессорные ЭВМ 263  
Многочлен 52, 56, 120  
— интерполяционный 51, 53  
— Лагранжа 53  
Модификация адреса 150

Нормализация числа 108, 133

Обработка изображений 39  
— информации 123  
— — текстовой 123, 224  
Обращение к подпрограмме 159  
Объем памяти 8, 82  
Однопроцессорные ЭВМ 263  
Округление чисел 43  
Операнд 84, 86  
Оператор 171, 176  
— безусловного перехода 182  
— ввода 207  
— вывода 207  
— присваивания 178  
— условного перехода 182  
— цикла 190  
Оператор DATA 179  
— DEF 201  
— DIM 199, 240, 256  
— DRAW 256  
— FOR 190  
— GOSUB 204  
— GOTO 182  
— IF 192  
— INPUT 207  
— MOVE 257  
— NEXT 192  
— NPLOT 256  
— OPEN 256  
— PLOT 251, 256  
— PRINT 209, 254  
— READ 179, 206  
— REM 206  
— RESTORE 181  
— RETURN 204  
— STRETCH 257  
— TURN 257  
— WEND 198  
— WHILE 197  
Операционная система 267  
Операция арифметическая 84, 139  
— логическая 84  
— присваивания 122

Основание системы счисления 92  
Отладка программы 169, 219  
Ошибки семантические 169  
— синтаксические 169, 219

Память ЭВМ 8, 79, 265, 271  
— — внешняя 80  
— — оперативная 8, 80  
Параметр подпрограммы 159  
— цикла 190  
Перевод чисел 96  
Переменная 174  
— простая 190, 202  
— с индексом 175  
Переполнение 273  
Персональные ЭВМ 9, 224, 267  
Перфокарта 6  
Плотность записи 83  
Подпрограмма 158, 203  
Погрешность 41  
— абсолютная 42  
— относительная 42  
Поколения ЭВМ 7, 8, 9, 262  
Последовательный доступ 83  
Поле адреса 87  
— операции 87  
Порядок числа 108, 113, 133;  
174, 273  
Предложение автокодной про-  
граммы 130  
Приближенное решение урав-  
нения 60  
Приближенные значения ве-  
личин 40, 69  
Приемы программирования 115  
Признак модификации 150  
Принцип хранимый програм-  
мы 7  
Программа 6, 8, 127  
Процесс алгоритмизации 32,  
33  
— алгоритмический 18  
— вычислительный 118  
— построения математической  
модели 32  
— циклический 118  
Процессор 79, 295  
— векторный 265  
— матричный 265

Разложение по степеням числа  
92, 94, 96  
Ранг символа 230  
Регистр адреса 149, 160

Регистр индикатора 275  
— команд 85  
— накапливающий 280  
— памяти 278  
— рабочий 275  
— сумматора 84  
— счетчика 85  
Редактирование программы 170  
— текста 224  
Режим мультидоступа 267  
— однопользовательский 267  
— работы ЭВМ  
— разделения времени 269  
— реального времени 267, 269.

СБИС 8, 262  
Семантика 169, 171  
Сеть локальная 265...  
— ЭВМ 265  
Символьное кодирование 127  
Синтаксис 169, 171  
Система алгоритмическая 20,  
22, 78  
— интерпретирующая 216  
— компилирующая 216  
— команд 87, 127, 132, 264  
— линейных алгебраических  
уравнений 44, 289  
— мультипроцессорная 266  
— программирования 267  
— счисления 92  
— восьмеричная 95, 101  
— — двоичная 93, 188  
— — двоично-восьмеричная 105  
— — двоично-десятичная 102  
— — двоично-шестнадцатерич-  
ная 105  
— — смешанная 102  
— — шестнадцатеричная 95  
Словесно-формульная запись  
алгоритма 124  
Смещение 91  
Стек 296  
Столбец 175  
Строка 175  
Сумматор 84  
Схема Горнера 25, 96, 120, 194  
— интегральная 8  
— электронная 8  
Счетчик команд 86, 88

Текст 225  
Текстовая константа 225  
— переменная 225

Тело цикла 191, 198  
Терминал 8, 85, 207  
Тест 225  
Тип операции 86  
Точность задания исходных данных 40  
— решения задачи 40, 122  
Транслятор 8, 129, 167  
Трансляция 217

Узел интерполирования 52, 54  
Уравнение алгебраическое 19  
— линейное 115  
Устройство арифметико-логическое 79, 84, 265  
— ввода 6, 79, 85  
— — изображений 36  
— вывода 6, 79  
— запоминающее 6, 79, 208  
— управления 6, 79, 84, 86, 289

Фактический аргумент 202, 205  
Фонд алгоритмов и программ 31  
Формальный аргумент 202, 205  
Формат команды 127  
Форматёр 243  
Формула квадратурная 69  
— Ньютона — Лейбница 68, 71  
— прямоугольников 69, 71  
— рекуррентная 64, 77, 98  
— трапеции 73  
Фортран 170  
Функция ТАВ 211, 215, 254  
— 213  
— 235  
— 240

Цикл 118, 120, 152, 186  
— вложенный 156, 189, 196  
— внешний 156  
— внутренний 156  
— с известным числом повторений 145, 148  
— с неизвестным числом повторений 145

Число нормализованное 108, 110  
— с двойной точностью 112  
— с плавающей запятой 107, 133, 173, 210, 273  
— с фиксированной запятой 107, 173, 210, 272

Эквивалентный алгоритм 120  
Экспертные системы 263  
Элементарный графический объект 250  
Элементная база 7  
Этапы алгоритмизации 34

Язык алгоритмический 8, 165, 171  
— алгоритмической системы 21  
— блок-схем 126  
— записи алгоритмов 78, 115  
— программирования 129  
— — машинно-независимый 129  
— — машинно-ориентированный 129  
— — символьный 129  
— — универсальный 170  
— ЭВМ 7, 8, 86, 115, 126  
Языка алфавит 171  
— семантика 171  
— синтаксис 171

*В л а с о в Виктор Константинович,  
К о р о л е в Лев Николаевич,  
С о т н и к о в Александр Николаевич*  
ЭЛЕМЕНТЫ ИНФОРМАТИКИ

С е р и я «Библиотечка программиста», вып. 52

Редактор *Е. В. Зима*

Художественный редактор *Г. М. Коровина*

Технический редактор *С. Я. Шкляр*

Корректоры *Г. В. Подвольская, М. Н. Дронова*

ИБ № 32355

Сдано в набор 23.07.87. Подписано к печати 02.06.88.  
Т-11696. Формат 84×108 1/32.

Бумага тип. № 3. Гарнитура литературная.

Печать высокая. Усл. печ. л. 16,8. Усл. кр.-отт. 17,01.

Уч.-изд. л. 20,43. Тираж 100 000 экз. Заказ № 8—363.

Цена 1 р. 30 к.

Ордена Трудового Красного Знамени

издательство «Наука»

Главная редакция физико-математической литературы

117071 Москва, В-71, Ленинский проспект, 15.

Набор и матрицы изготовлены в ордена Октябрьской  
Революции и ордена Трудового Красного Знамени  
МПО «Первая Образцовая типография» им. А. Жданова  
Союзполиграфпрома при Госкомиздате СССР.  
113054 Москва, Валовая, 28.

Отпечатано на полиграфкомбинате ЦК ЛКСМУ  
«Молодь» ордена Трудового Красного Знамени ИПО  
ЦК ВЛКСМ «Молодая гвардия». 252119. Киев,  
Пархоменко, 38—44.



Михаил Михайлович

С. 10